Inclui CD-ROM

SISTEMAS OPERACIONAIS

PROJETO E IMPLEMENTAÇÃO

Andrew S. Tanenbaum
Albert S. Woodhull

2ª Edição

Bookman

T164s Tanenbaum, Andrew S.
Sistemas operacionais: projeto e implementação / Andrew S. Tanenbaum e Albert S. Woodhull; trad. Edson Furmankiewicz. — 2. ed. — Porto Alegre: Bookman, 2000.

1. Administração — Sistemas operacionais. I. Woodhull, Albert S. II. Título

CDU 65.012

Catalogação na publicação: Mônica Ballejo Canto — CRB 10/1023

ISBN 85-7307-530-9

SISTEMAS OPERACIONAIS

PROJETO E IMPLEMENTAÇÃO

Andrew S. Tanenbaum
Vrije Universiteit, Amsterdã, Holanda
Albert S. Woodhull
Hampshire College, Amherst, Massachusetts, Estados Unidos

2ª Edição

Tradução:
EDSON FURMANKIEWICZ

Consultoria, supervisão e revisão técnica desta edição:
FABIAN L. BECKER
Sócio-gerente da Creare Informática Ltda.
FLÁVIO PIZZATO
Engenheiro eletrônico pela UFRGS.
Sócio-gerente da Produttare Consultores Associados

Reimpressão 2002

PORTO ALEGRE / 1999

Para Suzanne e para Barbara, Marvin e o pequeno Bram.

— AST

Para Barbara e para Gordon.

— ASW

Sobre os Autores

Andrew S. Tanenbaum é bacharel em Ciências pelo M.I.T. e Ph.D. pela California University, em Berkeley. Atualmente, é professor de Ciência da Computação na Vrije Universiteit em Amsterdã, Holanda, onde lidera o Grupo de Sistemas de Computação. O autor também é diretor da Faculdade Avançada de Computação e Imagens Digitais, uma instituição interuniversitária que faz pesquisa sobre sistemas avançados paralelos, distribuídos e de imagem. Contudo, está tentando evitar a todo custo transformar-se em um burocrata.

No passado, fez pesquisa sobre compiladores, sistemas operacionais, redes remotas e sistemas distribuídos em rede local. Suas pesquisas atuais concentram-se principalmente no projeto de sistemas distribuídos de rede remota que atingem a escala de milhões de usuários. Esses projetos de pesquisa levaram a mais de 70 artigos em periódicos e conferências, além de cinco livros.

O professor Tanenbaum também produziu um considerável volume de software. Ele foi o principal arquiteto do Amsterdam Compiler Kit, um *kit* de ferramentas largamente utilizado para escrever compiladores portáveis, assim como do MINIX. Junto com seus programadores e alunos Ph.D., ele ajudou a projetar o sistema operacional distribuído Amoeba, um sistema operacional distribuído de alto desempenho baseado em *microkernel*. O MINIX e o Amoeba estão agora disponíveis gratuitamente para fins educacionais e de pesquisa via Internet.

O professor Tanenbaum é *Fellow* da ACM, membro sênior do IEEE, membro da Real Academia de Artes e Ciências da Holanda, vencedor do Prêmio Karl V. Karlstrom de Destaque na Educação de 1994 da ACM e do Prêmio ACM/SIGCSE para Contribuições Destacadas para Educação da Ciência da Computação de 1997. Seu nome também consta no *Who's Who in the World*. Sua *home page* na World Wide Web pode ser localizada na URL *http: //www.cs.vu.nl/~ast/*

Albert S. Woodhull é bacharel em Ciências pelo M.I.T. e Ph.D. pela Washington University. Ele ingressou no M.I.T. com o objetivo de tornar-se engenheiro elétrico, mas saiu de lá biólogo. É professor associado da Escola de Ciência Natural do Hampshire College em Amherst, Massachusetts, desde 1973. Como biólogo, utilizando instrumentação eletrônica, começou a trabalhar com microcomputadores logo que estes apareceram. Seus cursos de instrumentação para alunos de ciências evoluíram para cursos sobre interfaces de computador e de programação em tempo real.

O Dr. Woodhull sempre teve forte interesse em ensinar e no papel da ciência e da tecnologia no desenvolvimento. Antes de entrar na faculdade, lecionou Ciência no nível secundário por dois anos na Nigéria. Mais recentemente passou vários fins-de-semana ensinando Ciência da Computação na Universidad Nacional de Ingenieria da Nicarágua e na Universidad Nacional Autónoma de Nicarágua.

Ele é interessado em computadores e em sistemas eletrônicos, bem como nas interações de computadores com outros sistemas eletrônicos. Gosta de ensinar nas áreas de arquitetura de computadores, programação em linguagem *assembly*, sistemas operacionais e comunicações de computador. Woodhull também trabalhou como consultor no desenvolvimento de instrumentação eletrônica e de software relacionado.

O Dr. Woodhull também tem muitos interesses não-acadêmicos, incluindo esportes ao ar livre, radioamadorismo e leitura. Ele gosta de viajar e tentar fazer-se entender em outros idiomas que não o seu inglês nativo. Sua *home page* na World Wide Web está localizada em um sistema executando MINIX, na URL *http: //minix1.hampshire.edu/asw/*.

Prefácio

A maioria dos livros sobre sistemas operacionais é rica na parte teórica e inconsistente, na prática. Este, contudo, objetiva proporcionar um melhor equilíbrio entre ambas. Ele abrange todos os princípios fundamentais pormenorizadamente, incluindo processos, comunicação interprocessos, semáforos, monitores, passagem de mensagens, algoritmos de agendamento, entrada/saída, impasses, *drivers* de dispositivos, gerenciamento de memória, algoritmos de paginação, projeto de sistema de arquivos, segurança e mecanismos de proteção, mas, também, discute em detalhes um sistema particular — o MINIX, sistema operacional compatível com o Unix — além de oferecer uma listagem completa do código-fonte para estudo. Esse arranjo permite que o leitor não só aprenda os princípios, mas, também, veja como são aplicados em um sistema operacional real.

Quando a primeira edição deste livro foi publicada, em 1987, causou uma pequena revolução na maneira como os cursos de sistemas operacionais eram ministrados. Até então, a maioria dos cursos somente abordava a teoria. Com o aparecimento do MINIX, muitas escolas começaram a oferecer aulas de laboratório, onde os alunos examinavam um sistema operacional real para ver como ele funcionava por dentro. Consideramos essa tendência muito bem-vinda e esperamos que esta segunda edição a fortaleça.

Nos seus primeiros 10 anos, o MINIX sofreu muitas mudanças. O código original foi projetado para um IBM PC baseado no 8088 de 256K com duas unidades de disquete e nenhum disco rígido. Além disso, foi baseado na Versão 7 do UNIX. Com o tempo, o MINIX desenvolveu-se de várias maneiras. Por exemplo, a versão atual agora rodará em qualquer computador situado entre o PC original (em modo real de 16 bits) até poderosos Pentium com grandes discos rígidos (no modo protegido de 32 bits). Ele também mudou da antiga base na Versão 7, para basear-se no padrão internacional POSIX (IEEE 1003.1 e ISO 9945-1). Por fim, muitos recursos foram acrescentados, demais em nosso ponto de vista, mas poucos, na visão de outras pessoas, que levaram à criação do LINUX. Além disso, o MINIX foi portado para muitas outras plataformas, para incluir o Macintosh, o Amiga, o Atari e as estações SPARC. Este livro abrange apenas o MINIX 2.0, que até agora só roda em computadores com uma CPU 80x86, em sistemas que podem emular uma CPU desse tipo ou em uma estação SPARC.

Esta segunda edição apresenta mudanças em todo o livro. Quase todo o material sobre princípios foi revisado, e considerável material novo foi acrescentado. Entretanto, a mudança principal é a discussão do novo MINIX baseado em POSIX, além da inclusão do novo código neste livro. Também nova é a inclusão de um CD-ROM, para oferecer o código-fonte completo do MINIX, com instruções para instalar o MINIX em um PC (veja o arquivo LEIAME.TXT no diretório principal do CD-ROM).

Instalar o MINIX em um PC 80x86, seja para uso individual seja para um laboratório, é simples e direto. Uma partição de disco de pelo menos 30MB deve ser feita para ele, que então pode ser instalado, bastando que as instruções do arquivo LEIAME.TXT do CD-ROM sejam seguidas. Para imprimir o arquivo LEIAME.TXT em um PC, primeiro inicie o MS-DOS, se ele ainda não estiver rodando (a partir do WINDOWS, clique no ícone MS-DOS). Então digite:

```
copy leiame.txt prn
```

para imprimir. O arquivo também pode ser examinado no Edit do MS-DOS, no WordPad, no Bloco de Notas do Windows, ou em qualquer outro editor de texto que manipule texto puro em ASCII.

Para escolas (ou indivíduos) que não dispõem de PCs, duas outras opções estão agora disponíveis. Foram incluídos dois simuladores no CD-ROM. Um, escrito por Paul Ashton, roda em SPARCs. Ele executa o MINIX como um programa de usuário sobre o Solaris. Como conseqüência, o MINIX é compilado em um binário SPARC e roda à toda velocidade. Deste modo, o MINIX não é mais um sistema

operacional, mas um programa de usuário. Portanto, foram necessárias algumas mudanças no código de baixo nível.

O outro simulador foi escrito por Kevin P. Lawton da Bochs Software Company. Esse simulador interpreta o conjunto de instruções Intel 80386 e o suficiente do mecanismo de E/S para o MINIX poder rodar no simulador. Naturalmente, rodar em um interpretador custa algum desempenho, mas facilita muito a depuração para o aluno. Esse simulador tem a vantagem de executar em qualquer computador que suporte o M.I.T. X Window System. Para mais informações sobre esses dois simuladores, veja o CD-ROM.

O desenvolvimento do MINIX é uma proposta em progresso. O conteúdo deste livro e seu CD-ROM são meramente uma amostra do sistema na época da sua publicação. Para o estado atual das coisas, visite a *home page* do MINIX na World Wide Web, *http://www.cs.vu.nl/~ast/minix.html*. Além disso, o MINIX tem seu próprio grupo de discussão na USENET: *comp.os.minix*, no qual os leitores podem inscrever-se para saber o que está acontecendo no mundo do MINIX. Para aqueles com correio eletrônico, mas sem acesso a grupos de discussão, também há uma lista de discussão via *e-mail*. Escreva para *listserv@listserv.nodak.edu* com "subscribe minix-l <seu nome completo>" como a primeira e única linha do corpo da mensagem. Você receberá mais informações pelo correio eletrônico.

Para uso em sala de aula, a Prentice-Hall oferece um manual de soluções de problemas, mas apenas para professores. Os arquivos PostScript com todas as figuras no livro, conveniente como material didático de apoio, podem ser encontradas seguindo o *link* "*Software and supplementary material*" em *http://www.cs.vu.nl/~ast/*.

Fomos felizes em contar com a ajuda de muitas pessoas durante o curso deste projeto. Antes de tudo, gostaríamos de agradecer a Kees Bot por fazer a parte do leão, no trabalho, adequando o MINIX ao padrão e gerenciando a distribuição. Sem sua grande ajuda nunca teríamos feito tudo isso. Ele escreveu pedaços grandes do código (p. ex., a E/S de terminal POSIX), limpou outras seções e corrigiu numerosos *bugs* que surgiram no decorrer dos anos.

Bruce Evans, Philip Homburg, Will Rose e Michael Temari contribuíram para o desenvolvimento do MINIX durante anos. Centenas de outras pessoas contribuíram para o MINIX por meio dos grupos de discussão. Há tantos deles e suas contribuições são tão variadas que não podemos nem começar a listar todos aqui. Então, o melhor que podemos fazer é dar um muito obrigado a todos eles.

Várias pessoas leram partes dos originais e apresentaram sugestões. Gostaríamos de dedicar nosso especial agradecimento a John Casey, Dale Grit e Frans Kaashoek.

Na Vrije Universiteit, diversos alunos testaram a versão beta do CD-ROM, destacando-se: Ahmed Batou, Goran Dokic, Peter Gijzel, Thomer Gil, Dennis Grimbergen, Roderick Groesbeek, Wouter Haring, Guido Kollerie, Mark Lassche, Raymond Ris, Frans ter Borg, Alex van Ballegooy, Ries van der Velden, Alexander Wels e Thomas Zeeman. Agradecemos pelo trabalho cuidadoso e pelos relatórios detalhados.

ASW também gostaria de agradecer a vários dos seus ex-alunos, particularmente Peter W. Young, do Hampshire College, e Maria Isabel Sanchez e William Puddy Vargas, da Universidad Nacional Autónoma de Nicarágua, cujo interesse no MINIX sustentou seus esforços.

Por fim, queremos agradecer às nossas famílias. Suzanne entrou nessa pela décima vez. Barbara pela nona vez. Marvin pela oitava. E até o pequeno Bram já está na sua quarta vez. A coisa parece estar tornando-se rotina, mas o amor e o apoio ainda muito são apreciados. (AST)

Quanto à Barbara, esta é a sua primeira vez. Nada teria sido possível sem seu apoio, paciência e bom humor. Foi sorte do Gordon ter estado longe, na faculdade, durante a maior parte disso tudo. Mas é maravilhoso ter um filho que entende e preocupa-se com as mesmas coisas que me fascinam. (ASW)

Andrew S. Tanenbaum
Albert S. Woodhull

Sumário

1 ASPECTOS GERAIS 17

1.1 O QUE É UM SISTEMA OPERACIONAL 18
1.1.1 O sistema operacional como uma máquina estendida 18
1.1.2 O sistema operacional como um gerenciador de recursos 19
1.2 HISTÓRIA DOS SISTEMAS OPERACIONAIS 19
1.2.1 A primeira geração (1945-55): válvulas e painéis de conectores 20
1.2.2 A segunda geração (1955-65): transístores e sistemas de lote 20
1.2.3 A terceira geração (1965-1980): CIs e multiprogramação 21
1.2.4 A quarta geração (1980-hoje): computadores pessoais 24
1.2.5 A história do MINIX 24
1.3 CONCEITOS DE SISTEMA OPERACIONAL 25
1.3.1 Processos 26
1.3.2 Arquivos 27
1.3.3 O *Shell* 29
1.4 CHAMADAS DE SISTEMA 30
1.4.1 Chamadas de sistema para gerenciamento de processos 30
1.4.2 Chamadas de sistema para sinalização 33
1.4.3 Chamadas de sistema para gerenciamento de arquivos 34
1.4.4 Chamadas de sistema para gerenciamento de diretórios 37
1.4.5 Chamadas de sistema para proteção 38
1.4.6 Chamadas de sistema para gerenciamento de tempo 39
1.5 A ESTRUTURA DO SISTEMA OPERACIONAL 39
1.5.1 Sistemas monolíticos 39
1.5.2 Sistemas em camadas 40
1.5.3 Máquinas virtuais 42
1.5.4 Modelo cliente-servidor 43
1.6 VISÃO GERAL DO RESTANTE DESTE LIVRO 44
1.7 RESUMO 44

2 PROCESSOS 47

2.1 INTRODUÇÃO AOS PROCESSOS 47
2.1.1 Modelo de processo 47
2.1.2 Implementação de processos 50
2.1.3 *Threads* 51
2.2 COMUNICAÇÃO INTERPROCESSO 53
2.2.1 Condições de corrida 53
2.2.2 Seções críticas 54
2.2.3 Exclusão mútua com espera ativa 54
2.2.4 *Sleep* e *wakeup* 57
2.2.5 Semáforos 58
2.2.6 Monitores 60
2.2.7 Passagem de mensagem 63
2.3 PROBLEMAS CLÁSSICOS DE CIP 64
2.3.1 O problema dos filósofos jantando 64
2.3.2 O problema dos leitores e dos escritores 66
2.3.3 O problema do barbeiro adormecido 66
2.4 AGENDAMENTO DE PROCESSO 69
2.4.1 Agendamento *round robin* 70
2.4.2 Agendamento por prioridade 71
2.4.3 Múltiplas filas 71
2.4.4 *Job* mais curto primeiro 72
2.4.5 Agendamento garantido 73
2.4.6 Agendamento por sorteio 73
2.4.7 Agendamento de tempo real 74
2.4.8 Agendamento de dois níveis 75
2.4.9 Política *versus* mecanismo 75
2.5 VISÃO GERAL DE PROCESSOS EM MINIX 76
2.5.1 A estrutura interna do MINIX 76
2.5.2 Gerenciamento de processos no MINIX 77
2.5.3 Comunicação interprocessos no MINIX 78
2.5.4 Agendamento de processos no MINIX 79
2.6 IMPLEMENTAÇÃO DE PROCESSOS EM MINIX 79
2.6.1 Organização do código-fonte do MINIX 79
2.6.2 Os arquivos de cabeçalho comuns 81
2.6.3 Arquivos de cabeçalho do MINIX 84
2.6.4 Estruturas de dados de processo e arquivos de cabeçalho 88
2.6.5 Fazendo a inicialização do MINIX 92
2.6.6 Inicialização do sistema 94
2.6.7 Tratamento de interrupções no MINIX 97
2.6.8 Comunicação interprocesso no MINIX 103
2.6.9 Agendamento no MINIX 105
2.6.10 Suporte de *kernel* dependente de hardware 106
2.6.11 Utilitários e biblioteca do *kernel* 108
2.7 RESUMO 109

3 ENTRADA/SAÍDA 113

3.1 PRINCÍPIOS DO HARDWARE DE E/S 113
3.1.1 Dispositivos de E/S 113
3.1.2 Controladoras de dispositivo 114
3.1.3 Acesso direto à memória (DMA) 115
3.2 PRINCÍPIOS BÁSICOS DO SOFTWARE DE E/S 117
3.2.1 Metas do software de E/S 117
3.2.2 Manipuladores de interrupções 118

3.2.3 *Drivers* de dispositivos 118
3.2.4 Software de E/S independente de dispositivo 119
3.2.5 Software de E/S no espaço do usuário 120
3.3 IMPASSES 121
3.3.1 Recursos 122
3.3.2 Princípios básicos de impasses 122
3.3.3 O algoritmo do avestruz 124
3.3.4 Detecção e recuperação 124
3.3.5 Prevenção de impasses 124
3.3.6 Impedimento de impasses 127
3.4 VISÃO GERAL DE E/S NO MINIX 130
3.4.1 Manipuladores de interrupções no MINIX 130
3.4.2 *Drivers* de dispositivos no MINIX 131
3.4.3 Software de E/S independente de dispositivo no MINIX 133
3.4.4 Software de E/S no nível do usuário no MINIX 134
3.4.5 Manipulação de impasses no MINIX 134
3.5 DISPOSITIVOS DE BLOCO NO MINIX 135
3.5.1 Visão geral de *drivers* de dispositivo de bloco no MINIX 135
3.5.2 Software comum de *driver* de dispositivo de bloco 136
3.5.3 A biblioteca de *driver* 139
3.6 DISCOS DE RAM 140
3.6.1 Hardware e software de disco de RAM 140
3.6.2 Visão geral do *driver* de disco de RAM no MINIX 141
3.6.3 Implementação do *driver* de disco de RAM no MINIX 142
3.7 DISCOS 143
3.7.1 Hardware de disco 143
3.7.2 Software de disco 144
3.7.3 Visão geral do *driver* de disco rígido no MINIX 148
3.7.4 A implementação do *driver* de disco rígido no MINIX 150
3.7.5 Manipulação de disquete 155
3.8 RELÓGIOS 156
3.8.1 Hardware do relógio 156
3.8.2 Software do relógio 157
3.8.3 Visão geral do *driver* de relógio no MINIX 159
3.8.4 Implementação do *driver* de relógio no MINIX 161
3.9 TERMINAIS 164
3.9.1 Hardware de terminal 164
3.9.2 Software de terminal 168
3.9.3 Visão geral do *driver* de terminal no MINIX 173
3.9.4 Implementação do *driver* de terminal independente de dispositivo 183
3.9.5 Implementação do *driver* de teclado 193
3.9.6 Implementação do *driver* de vídeo 197
3.10 A TAREFA DO SISTEMA NO MINIX 202
3.11 RESUMO 207

4 GERENCIAMENTO DE MEMÓRIA 211

4.1 GERENCIAMENTO BÁSICO DE MEMÓRIA 211
4.1.1 Monoprogramação sem troca ou paginação 212
4.1.2 Multiprogramação com partições fixas 212
4.2 TROCA 214
4.2.1 Gerenciamento de memória com mapas de bits 215
4.2.2 Gerenciamento de memória com listas encadeadas 216
4.3 MEMÓRIA VIRTUAL 217
4.3.1 Paginação 218
4.3.2 Tabelas de página 220

4.3.3 TLBs — *Translation Lookaside Buffers* 223
4.3.4 Tabelas de páginas invertidas 224
4.4 ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINA 225
4.4.1 Algoritmo de substituição de página ótimo 225
4.4.2 Algoritmo de substituição de página não recentemente utilizada 226
4.4.3 Algoritmo de substituição de página primeiro a entrar, primeiro a sair (FIFO) 226
4.4.4 Algoritmo de substituição de página de segunda chance 227
4.4.5 Algoritmo de substituição de página do relógio 227
4.4.6 Algoritmo de substituição de página menos recentemente utilizada (LRU) 228
4.4.7 Simulação de LRU em software 228
4.5 QUESTÕES DE PROJETO PARA SISTEMAS DE PAGINAÇÃO 229
4.5.1 Modelo de conjunto funcional 230
4.5.2 Política de alocação local *versus* global 231
4.5.3 Tamanho de página 232
4.5.4 Interface de memória virtual 233
4.6 SEGMENTAÇÃO 234
4.6.1 Implementação da segmentação pura 236
4.6.2 Segmentação com paginação: MULTICS 236
4.6.3 Segmentação com paginação: o Pentium Intel 238
4.7 VISÃO GERAL DO GERENCIAMENTO DE MEMÓRIA NO MINIX 243
4.7.1 Leiaute de memória 244
4.7.2 Processamento de mensagens 246
4.7.3 Estrutura de dados e algoritmos do gerenciador de memória 246
4.7.4 Chamadas de sistema FORK, EXIT e WAIT 250
4.7.5 Chamada de sistema EXEC 251
4.7.6 Chamada de sistema de BRK 253
4.7.7 Manipulação de sinais 253
4.7.8 Outras chamadas de sistema 257
4.8 IMPLEMENTAÇÃO DO GERENCIAMENTO DE MEMÓRIA NO MINIX 258
4.8.1 Arquivos de cabeçalho e estruturas de dados 258
4.8.2 Programa principal 259
4.8.3 Implementação de FORK, EXIT e WAIT 260
4.8.4 Implementação de EXEC 261
4.8.5 Implementação de BRK 262
4.8.6 Implementação da manipulação de sinais 263
4.8.7 Implementação das outras chamadas de sistema 267
4.8.8 Utilitários do gerenciador de memória 267
4.9 RESUMO 268

5 SISTEMAS DE ARQUIVOS 271

5.1 ARQUIVOS 271
5.1.1 Nomes de arquivo 272
5.1.2 Estrutura de arquivos 272
5.1.3 Tipos de arquivo 273
5.1.4 Acesso a arquivos 275
5.1.5 Atributos de arquivo 275
5.1.6 Operações com arquivos 276
5.2 DIRETÓRIOS 277
5.2.1 Sistemas hierárquicos de diretórios 277
5.2.2 Nomes de caminho 278
5.2.3 Operações com diretórios 280
5.3 IMPLEMENTAÇÃO DO SISTEMA DE ARQUIVOS 280
5.3.1 Implementando arquivos 280
5.3.2 Implementando diretórios 282

5.3.3 Gerenciamento de espaço em disco 284
5.3.4 Confiabilidade do sistema de arquivos 287
5.3.5 Desempenho do sistema de arquivos 290
5.3.6 Sistema de arquivos estruturados em *log* 291
5.4 SEGURANÇA 293
5.4.1 Ambiente de segurança 293
5.4.2 Falhas famosas de segurança 294
5.4.3 Ataques de segurança genéricos 296
5.4.4 Princípios de projeto para segurança 297
5.4.5 Autenticação do usuário 297
5.5 MECANISMOS DE PROTEÇÃO 300
5.5.1 Domínios de proteção 300
5.5.2 Listas de controle de acesso 301
5.5.3 Capacidades 302
5.5.4 Canais secretos 303
5.6 VISÃO GERAL DO SISTEMA DE ARQUIVOS DO MINIX 304
5.6.1 Mensagens 305
5.6.2 Arranjo do sistema de arquivos 305
5.6.3 Mapas de bits 308
5.6.4 Nós-i 309
5.6.5 *Cache* de blocos 309
5.6.6 Diretórios e caminhos 311
5.6.7 Descritores de arquivos 312
5.6.8 Bloqueio de arquivos 313
5.6.9 Canalizações e arquivos especiais 314
5.6.10 Um exemplo: chamada de sistema READ 315
5.7 IMPLEMENTAÇÃO DO SISTEMA DE ARQUIVOS DO MINIX 315
5.7.1 Arquivos de cabeçalho e estruturas de dados globais 315
5.7.2 Gerenciamento de tabelas 318
5.7.3 Programa principal 323
5.7.4 Operações em arquivos individuais 325
5.7.5 Diretórios e caminhos 330
5.7.6 Outras chamadas de sistema 332
5.7.7 Interface de dispositivos de E/S 335
5.7.8 Utilitários gerais 335
5.8 RESUMO 336

6 LISTA DE LEITURA E BIBLIOGRAFIA 339

6.1 SUGESTÕES PARA LEITURAS SUPLEMENTARES 339
6.1.1 Introdução e trabalhos gerais 339
6.1.2 Processos 340
6.1.3 Entrada/Saída 340
6.1.4 Gerenciamento de memória 341
6.1.5 Sistema de arquivos 341
6.2 BIBLIOGRAFIA 341

APÊNDICES

A O CÓDIGO-FONTE DO MINIX 347

B ÍNDICE PARA OS ARQUIVOS 731

C ÍNDICE PARA OS SÍMBOLOS 735

ÍNDICE 751

1

Aspectos Gerais

Sem software, um computador é basicamente um inútil amontoado de metal. Com software, um computador pode armazenar, processar e recuperar informações, exibir documentos de multimídia, pesquisar na Internet e envolver-se em muitas outras importantes atividades que justificam seu valor. O software de computador pode ser dividido, grosso modo, em duas espécies: programas de sistema, que gerenciam a operação do computador em si, e programas aplicativos, que executam o trabalho que o usuário realmente deseja. O programa de sistema mais fundamental é o **sistema operacional**, que controla todos os recursos do computador e fornece a base sobre a qual os programas aplicativos podem ser escritos.

Um moderno sistema de computador consiste em um ou mais processadores, alguma memória principal (também conhecida como RAM — *Random Access Memory*, Memória de Acesso Aleatório), discos, impressoras, interfaces de rede e outros dispositivos de entrada/saída. Em suma, um sistema complexo. Escrever os programas que controlam todos esses componentes e usá-los corretamente é um trabalho extremamente difícil. Se cada programador tivesse de preocupar-se com o modo como as unidades de disco funcionam e com todas as dúzias de coisas que poderiam dar errado ao ler um bloco de disco, seria provável que muitos programas sequer pudessem ser escritos.

Há muitos anos tornou-se bastante evidente a necessidade de encontrar uma maneira de isolar os programadores da complexidade do hardware. A maneira com que isso se desenvolveu gradualmente foi colocar uma camada de software por cima do hardware básico para gerenciar todas as partes do sistema e oferecer ao usuário uma interface ou **máquina virtual** que é mais fácil de entender e de programar. Essa camada de software é o sistema operacional e constitui o assunto deste livro.

A situação é mostrada na Figura 1-1. Na parte inferior, está o hardware, que, em muitos casos, é composto de duas ou mais camadas. A camada baixa contém dispositivos físicos, consistindo em circuitos integrados, cabos, fonte alimentadora, tubos de raios catódicos e dispositivos físicos semelhantes. Como tais elementos são construídos e como funcionam é assunto do engenheiro elétrico.

Em seguida (em algumas máquinas), vem uma camada de software primitivo que controla diretamente esses dispositivos e proporciona uma interface limpa para a próxima camada. Esse software, chamado **microprograma**, normalmente está localizado em memória somente para leitura. Ele é realmente um interpretador, buscando as instruções de linguagem de máquina como ADD, MOVE e JUMP, executando-as como uma série de pequenos passos. Para executar uma instrução ADD, por exemplo, o microprograma deve determinar onde os números a serem somados estão localizados, buscá-los, adicioná-los e armazenar o resultado em algum lugar. O conjunto de instruções que o microprograma interpreta define a **linguagem de máquina**, que não é realmente parte do hardware da máquina, mas os fabricantes de computador sempre a descrevem em seus manuais dessa maneira, de modo que as pessoas pensam que ela é a "máquina" real.

Alguns computadores, chamados de máquinas **RISC (*Reduced Instruction Set Computers*)**, não têm um nível de microprogramação. Nestas máquinas, o hardware executa diretamente as instruções da linguagem de máquina. Como exemplos, o Motorola 680x0 tem um nível de microprogramação, mas o IBM PowerPC não.

A linguagem de máquina tipicamente tem entre 50 e 300 instruções que, na sua maior parte, servem para mover dados pela máquina, para fazer aritmética e para comparar valores. Nessa camada, os dispositivos de entrada/saída são controlados carregando valores em registradores especiais de dispositivo. Por exemplo, um disco pode ser comandado para ler carregando os valores do endereço do disco, o endereço de memória principal, a contagem de

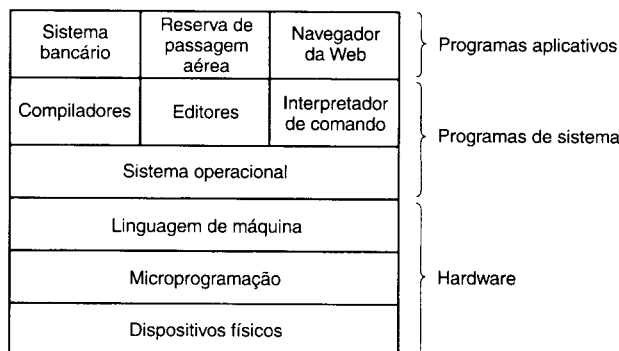



Figura 1-1 Um sistema de computador consiste em hardware, em programas de sistema e em programas aplicativos.

bytes e a instrução (READ ou WRITE) em seus registradores. Na prática, muitos parâmetros a mais são necessários e o resultado retornado pela unidade depois de uma operação é altamente complexo. Além disso, para muitos dispositivos de E/S, a temporizanação desempenha um papel importante na programação.

Uma importante função do sistema operacional é esconder toda essa complexidade e oferecer um conjunto mais conveniente de instruções para o programador trabalhar. Por exemplo, READ BLOCK FROM FILE é conceitualmente mais simples que ter de preocupar-se com os detalhes do movimento das cabeças de disco, esperar que elas abaixem e assim por diante.

Por cima do sistema operacional está o resto do software de sistema. Aqui encontramos o interpretador de comandos (*shell*), sistemas de janelas, compiladores, editores e programas independentes de aplicação semelhantes. É importante saber que esses programas definitivamente não são parte do sistema operacional, mesmo que eles tipicamente sejam fornecidos pelo fabricante do computador. Esse é um ponto crucial, mas sutil. O sistema operacional é aquela porção do software que executa no **modo *kernel*** ou no **modo de supervisor.** Ele é protegido do usuário pelo hardware (ignorando por enquanto os microprocessadores mais antigos que não tinham nenhuma proteção de hardware). Os compiladores e editores executam no **modo de usuário.** Se um usuário não gosta de um compilador particular, ele é livre para escrever seu próprio compilador se preferir; mas ele não é livre para escrever seu próprio manipulador de interrupções de disco, que é parte do sistema operacional e normalmente é protegido por hardware contra tentativas dos usuários de modificá-lo.

Por fim, acima dos programas de sistema vêm os programas aplicativos. Esses programas são comprados ou são escritos pelos usuários para resolver seus problemas particulares, como processadores de texto, planilhas eletrônicas, programas de cálculo de engenharia ou jogos.

1.1 O QUE É UM SISTEMA OPERACIONAL?

A maioria dos usuários de computador teve alguma experiência com um sistema operacional, embora seja difícil precisar exatamente o que é um sistema operacional. Parte do problema é que os sistemas operacionais executam basicamente duas funções não-relacionadas e dependendo de quem está falando, você ouve mais uma coisa ou outra. Vamos examinar as duas agora.

1.1.1 O Sistema Operacional como uma Máquina Estendida

Como mencionado anteriormente, a **arquitetura** (conjunto de instruções, organização da memória, E/S e estrutura de barramento) da maior parte dos computadores no nível da linguagem de máquina é primitiva e desajeitada para programar, especialmente para entrada/saída. Para tornar essa idéia mais palpável, veja como a E/S de disquete é feita usando o chip controlador NEC PD765 (ou equivalente), utilizado na maioria dos computadores pessoais.

O PD765 tem 16 comandos, cada um especificado carregando-se entre 1 e 9 bytes em um registrador de dispositivo. Esses comandos são para ler e para gravar dados, para mover o braço de disco e para formatar trilhas, assim como para inicializar, avaliar, ressetar e recalibrar a controladora e as unidades.

Os comandos mais básicos são READ e WRITE, cada um deles requerendo 13 parâmetros, compactados em 9 bytes. Tais parâmetros especificam itens como o endereço do bloco de disco a ser lido, o número de setores por trilha, o modo de gravação utilizado no meio físico, o tamanho do intervalo entre setores e o que fazer com uma marca de endereço de dados excluídos. Se você não entende este palavrório, não se preocupe; é essa exatamente a questão — é tudo muito esotérico. Quando a operação é completada,

o chip controlador retorna 23 campos de status e erro compactados em 7 bytes. Como se isso não fosse suficiente, o programador do disquete também deve estar constantemente ciente sobre se o motor está ligado ou desligado. Se o motor estiver desligado, ele deve ser ligado (com uma demora longa de inicialização) antes de os dados poderem ser lidos ou gravados. Mas o motor não pode permanecer ligado por muito tempo, senão o disquete irá desgastar-se. O programador assim é forçado a negociar entre a demora na inicialização e o desgaste dos disquetes (e a perda dos dados neles).

Sem entrar nos detalhes *reais*, deve estar claro que o programador médio provavelmente não quer ficar muito intimamente envolvido com a programação de disquetes (nem discos rígidos, que são igualmente complexos, apesar de bem diferentes). Em vez disso, o que o programador quer é uma abstração de ordem superior mais simples de lidar. No caso dos discos, uma abstração típica seria que o disco contém uma coleção de nomes de arquivos. Cada arquivo pode ser aberto para leitura ou gravação, então lido ou gravado e finalmente fechado. Detalhes como se a gravação deve ou não usar modulação de freqüência modificada e qual é o estado atual do motor não devem aparecer na abstração apresentada ao usuário.

O programa que esconde do programador a verdade sobre o hardware e apresenta uma bela e simples visão de nomes de arquivos que podem ser lidos e gravados é, naturalmente, o sistema operacional. Assim como o sistema operacional esconde do programador o hardware de disco e apresenta uma interface orientada para arquivo mais simples, ele também oculta muitas coisas desagradáveis relacionadas com interrupções, temporizadores, gerenciamento de memória e outros recursos de baixo nível. Em cada caso, a abstração oferecida pelo sistema operacional é mais simples e mais fácil de utilizar que o hardware subjacente.

Deste ponto de vista, a função do sistema operacional é apresentar ao usuário o equivalente de uma **máquina estendida** ou **máquina virtual** que é mais fácil de programar que o hardware subjacente. O modo como o sistema operacional alcança esse objetivo é uma longa história, que estudaremos em detalhe ao longo de todo este livro.

1.1.2 O Sistema Operacional como um Gerenciador de Recursos

O conceito do sistema operacional como principalmente oferecendo aos seus usuários uma interface conveniente é uma visão *top-down**. Uma visão alternativa, *botton-up***, sustenta que o sistema operacional está aí para gerenciar todas as partes de um sistema complexo. Os computadores modernos consistem em processadores, memórias, temporizadores, discos, mouses, interfaces de rede, impressoras a *laser* e uma ampla variedade de outros dispositivos. Na visão alternativa, o trabalho do sistema operacional é oferecer uma alocação ordenada e controlada dos processadores, das memórias e dos dispositivos de E/S entre os vários programas que competem por eles.

Imagine o que aconteceria se três programas que executam em algum computador tentassem imprimir simultaneamente na mesma impressora. As primeiras poucas linhas de impressão talvez sejam do programa 1, as poucas linhas seguintes do programa 2, então algumas do programa 3, etc. O resultado seria o caos. O sistema operacional pode trazer ordem a esse caos potencial armazenando toda a saída destinada para a impressora no disco. Quando um programa tiver terminado, o sistema operacional pode, então, copiar sua saída a partir do arquivo de disco onde ela foi armazenada para a impressora, enquanto, ao mesmo tempo, o outro programa pode continuar a gerar mais saída, ignorando o fato de que a saída realmente não está indo para a impressora (ainda).

Quando um computador (ou uma rede) tem múltiplos usuários, a necessidade de gerenciar e de proteger a memória, os dispositivos de E/S e outros recursos é ainda maior, uma vez que os usuários talvez interfiram um com outro. Além disso, os usuários freqüentemente necessitam não só compartilhar hardware, mas também as informações (arquivos, bases de dados, etc.). Em resumo, essa visão do sistema operacional sustenta que sua tarefa primária é monitorar quem está utilizando qual recurso, atender requisições de recurso, medir a utilização dos recursos e medir as requisições conflitantes de diferentes programas e usuários.

1.2 A HISTÓRIA DOS SISTEMAS OPERACIONAIS

Os sistemas operacionais vêm desenvolvendo-se ao longo dos anos. Nas próximas seções, veremos resumidamente esse desenvolvimento. Uma vez que os sistemas operacionais historicamente estiveram intimamente associados à arquitetura dos computadores nos quais eles rodam, examinaremos as sucessivas gerações de computadores para ver como eram seus sistemas operacionais. Tal relacionamento das gerações de sistemas operacionais com as gerações de computadores é grosseiro, mas oferece uma base que de outra forma não teríamos.

O primeiro computador digital verdadeiro foi projetado pelo matemático inglês Charles Babbage (1792-1871). Embora Babbage tenha gasto a maior parte de sua vida e sua fortuna tentando construir seu "motor analítico", ele nunca conseguiu fazê-lo funcionar adequadamente porque a coisa era puramente mecânica e a tecnologia do seu tempo não podia produzir as necessárias rodas e engrenagens de alta precisão de que ele precisava. É desnecessário dizer, mas o motor analítico não tinha um sistema operacional.

Como um interessante dado histórico, Babbage sabia que precisaria de um software para seu motor analítico, assim ele contratou uma jovem mulher, chamada Ada Lovelace,

*N. de T. *Top-down*: visão de cima para baixo.
**N. de T. *Botton-up*: visão de baixo para cima.

que era filha do famoso poeta britânico Lord Byron, como primeiro programador do mundo. O nome da linguagem de programação Ada® foi criado em sua homenagem.

1.2.1 A Primeira Geração (1945-55): Válvulas e Painéis de Conectores

Depois dos esforços malsucedidos de Babbage, pouco progresso foi alcançado na construção de computadores digitais até a II Guerra Mundial. Em meados da década de 40, Howard Aiken, de Harvard, John von Neumann, do Instituto de Estudos Avançados de Princeton, J. Presper Eckert e William Mauchley, da Universidade da Pensilvânia, e Konrad Zuse, na Alemanha, entre outros, tiveram sucesso na construção de máquinas de cálculo utilizando válvulas. Essas máquinas eram enormes, ocupando salas inteiras com dezenas de milhares de válvulas, e eram muito mais lentas que as mais simples calculadoras de hoje.

Naqueles tempos, um único grupo de pessoas projetava, construía, programava, operava e mantinha cada máquina. Toda a programação era feita em linguagem de máquina pura, freqüentemente ligando com fios painéis de conectores para controlar as funções básicas da máquina. As linguagens de programação eram desconhecidas (nem mesmo a linguagem *assembly*) e ninguém nunca tinha ouvido falar de sistemas operacionais. O modo normal de operação era o programador reservar um período de tempo na folha de reserva na parede, depois descer até o lugar da máquina, inserir suas conexões no computador e gastar algumas horas esperando que nenhuma das aproximadamente 20.000 válvulas queimasse durante a execução. Praticamente todos os problemas eram simples cálculos numéricos, tais como gerar tabelas de senos e de co-senos.

No início da década de 50, a rotina havia melhorado um pouco com a introdução dos cartões perfurados. Agora era possível gravar programas em cartões e lê-los, em vez de usar cabos e conectores; fora isso, o procedimento era o mesmo.

1.2.2 A Segunda Geração (1955-65): Transistores e Sistemas de Lote

A introdução do transistor, em meados da década de 50, mudou o quadro radicalmente. Os computadores tornaram-se confiáveis o bastante para serem fabricados e vendidos para clientes com a expectativa de que continuariam a funcionar por tempo suficiente para realizar algum trabalho útil. Pela primeira vez, havia uma separação clara entre projetistas, construtores, operadores, programadores e o pessoal da manutenção.

Essas máquinas eram guardadas em salas especiais com ar-condicionado e equipes de operadores profissionais para mantê-las funcionando. Somente as grandes corporações ou importantes órgãos do governo ou universidades podiam ter recursos para arcar com seu preço, na casa dos milhões de dólares. Para executar um *job* (i. e., um programa ou um conjunto de programas), um programador primeiro escrevia o programa em papel (em *FORTRAN* ou *assembly*), e então o transformava em cartões perfurados. Ele então levava o conjunto de cartões para a sala de entrada e entregava-os a um dos operadores.

Quando o computador acabava o *job* que estava executando, um operador ia até a impressora, removia a saída e a levava para a sala de saída, para que o programador pudesse recolhê-la mais tarde. Então, o operador pegaria um dos conjuntos de cartões que haviam sido trazidos da sala de entrada e os inseria na máquina para leitura. Se o compilador *FORTRAN* fosse necessário, o operador teria que pegá-lo em um gabinete de arquivos e inseri-lo para leitura. Muito tempo de máquina era desperdiçado enquanto os operadores andavam em volta do computador.

Dado o alto custo do equipamento, não é de surpreender que as pessoas rapidamente começassem a procurar maneiras de reduzir o tempo desperdiçado. A solução geralmente adotada era o **sistema de processamento em lotes** (ou *batch system*). A idéia subjacente a essa solução era colecionar uma bandeja completa de *jobs* na sala de entrada e então lê-los sobre uma fita magnética utilizando um computador pequeno (relativamente) e barato como o IBM 1401, que era muito bom para ler cartões, para copiar fitas e para imprimir a saída, mas péssimo em cálculos numéricos. Outras máquinas muito mais caras, como o IBM 7094, eram utilizadas para a computação de fato. Essa situação é mostrada na Figura 1-2.

Depois de cerca de uma hora de coleta de um lote de *jobs*, a fita era rebobinada e levada à sala de máquinas, onde era montada em uma unidade de fita. O operador então, carregava um programa especial (o antepassado do sistema operacional de hoje), que lia o primeiro *job* da fita e executava. A saída era gravada em uma segunda fita em vez de ser impressa. Depois que cada *job* acabava, o sistema operacional automaticamente lia o próximo *job* da fita e começava a executá-lo. Quando o lote inteiro estava completo, o operador removia as fitas de entrada e saída, substituía a fita de entrada pelo próximo lote e levava a fita de saída para um 1401 imprimir *off line* (i. e., não-conectado ao computador principal).

A estrutura de um típico *job* de entrada é mostrada na Figura 1-3. Ela começava com um cartão $JOB, especificando o tempo máximo de execução em minutos, o número de conta a ser carregado e o nome do programador. Então, vinha um cartão $FORTRAN, instruindo o sistema operacional a carregar o compilador FORTRAN da fita de sistema, o qual era seguido pelo programa a ser compilado e então um cartão $LOAD, orientando o sistema operacional a carregar o programa objeto recém-compilado. (Programas compilados freqüentemente eram escritos em fitas virgens e tinham de ser carregados explicitamente.) Em seguida, vinha o cartão $RUN, instruindo o sistema operacional a executar o programa com os dados que o seguiam. Finalmente, o cartão $END marcava o fim do *job*. Esses cartões primitivos de controle foram os precursores das linguagens modernas de controle de *jobs* e interpretadores de comando.

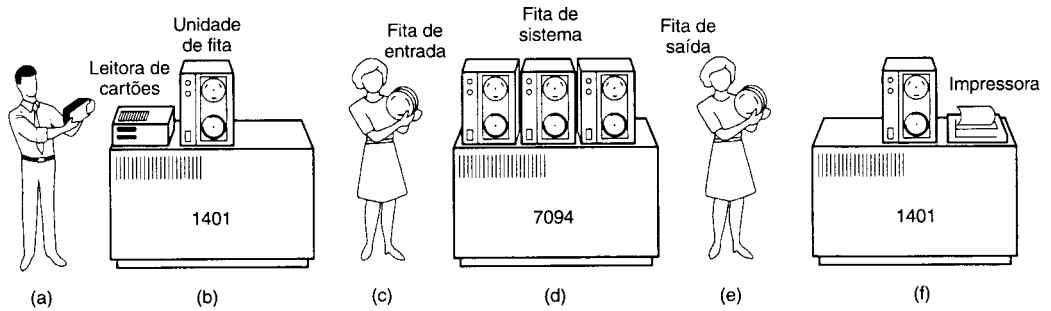



Figura 1-2 Um sistema de lote primitivo. (a) Programadores trazem os cartões para o 1401. (b) O 1401 lê os *jobs* em lote na fita. (c) O operador leva a fita de entrada para o 7094. (d) O 7094 realiza a computação. (e) O operador leva a fita de saída para o 1401. (f) O 1401 imprime a saída.

Grandes computadores de segunda geração eram utilizados principalmente para cálculos científicos de engenharia, tal como resolver equações diferenciais parciais. Eles em grande parte eram programados em FORTRAN e linguagem *assembler*. Sistemas operacionais típicos eram o FMS (o *Fortran Monitor System*) e o IBSYS, sistema operacional da IBM para o 7094.

1.2.3 A Terceira Geração (1965-1980): CIs e Multiprogramação

No início da década de 60, a maioria dos fabricantes de computadores tinha duas linhas de produto distintas e totalmente incompatíveis. De um lado, havia os computadores científicos de grande escala, baseados em palavras, como o 7094, que eram utilizados para cálculos numéricos em Ciência e em Engenharia. Por outro lado, havia os computadores comerciais, baseados em caracteres, como o 1401, que eram amplamente utilizados para classificar e para imprimir fitas para bancos e para companhias de seguro.

Desenvolver e manter duas linhas completamente diferentes de produtos era uma proposta cara para os fabricantes. Além disso, muitos novos clientes necessitavam, inicialmente, de uma máquina pequena. Mais tarde cresciam e queriam uma máquina maior que executasse todos os seus antigos programas, só que mais rapidamente.

A IBM tentou resolver ambos esses problemas em uma única tacada introduzindo o System/360. O 360 era uma série de máquinas compatíveis ao nível de software que variavam desde a capacidade de um 1401 até um muito mais

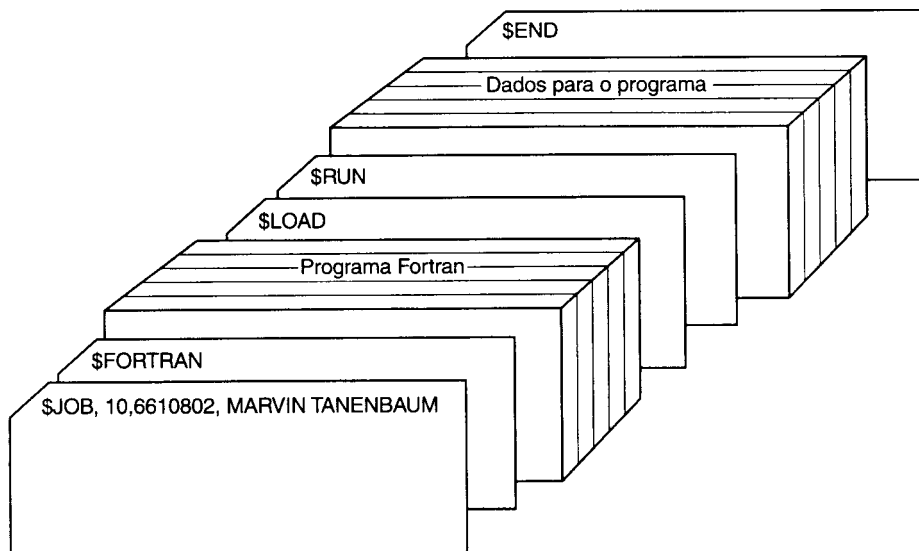



Figura 1-3 A estrutura de um típico *job* do FMS.

poderoso que o 7094. As máquinas diferiam só no preço e no desempenho (memória máxima, velocidade de processador, número de dispositivos de E/S permitidos, etc.). Como todas as máquinas tinham a mesma arquitetura e conjunto de instruções, programas escritos para uma máquina podiam executar em todas as outras, pelo menos na teoria. Além disso, o 360 foi projetado para manipular cálculos tanto científicos como comerciais. Assim, uma única família de máquinas podia satisfazer as necessidades de todos clientes. Nos anos seguintes, a IBM lançou sucessores compatíveis com a linha 360, usando tecnologia mais moderna, conhecidos como as séries 370, 4300, 3080 e 3090.

O 360 era a primeira linha importante de computadores usando Circuitos Integrados (CIs) — de pequena escala —, proporcionando assim uma vantagem importante de preço/desempenho sobre as máquinas de segunda geração, que eram baseadas em transistores como componentes. Ele foi um êxito imediato e a idéia de uma família de computadores compatíveis logo foi adotada por todos os outros fabricantes importantes. Os descendentes dessas máquinas ainda estão em uso hoje em centros de computação espalhados por aí, mas seu uso está declinando rapidamente.

A maior força da idéia de "uma família" era ao mesmo tempo sua maior fraqueza. A intenção era que todo software, incluindo o sistema operacional, tinha de trabalhar em todos os modelos. Ele tinha de executar em sistemas pequenos, que freqüentemente apenas substituíam os 1401 para copiar cartões em fita, e em sistemas muito grandes, que freqüentemente substituíam os 7094 para fazer previsão do tempo e outros cálculos pesados. Ele tinha de ser bom em sistemas com poucos periféricos e em sistemas com muitos periféricos. Ele tinha de funcionar em ambientes comerciais e em ambientes científicos. Acima de tudo, ele tinha de ser eficiente para todos esses diferentes usos.

Não havia como a IBM (ou qualquer outra pessoa) conseguir escrever um software para atender a todos esses requisitos contraditórios. O resultado era um sistema operacional extraordinariamente complexo e grande, provavelmente duas a três ordens de magnitude maior que o FMS. Ele consistia em milhões de linhas de linguagem *assembler* escritas por milhares de programadores com milhares e milhares de *bugs**, que necessitavam de um contínuo fluxo de novas versões na tentativa de corrigi-los. Cada nova versão corrigia alguns *bugs* e introduzia novos, então o número de *bugs* provavelmente permanecia constante com o tempo.

Um dos projetistas do OS/360, Fred Brooks, posteriormente escreveu um livro incisivo e perspicaz (Brooks, 1975) descrevendo suas experiências com o OS/360. Embora seja impossível resumir o livro aqui, basta dizer que a capa mostra uma horda de bestas pré-históricas atoladas em uma poça de petróleo. A capa do livro de Silberschatz e Galvin (1994) faz uma comparação semelhante.

Apesar do seu enorme tamanho e de seus problemas, o OS/360 e os sistemas operacionais semelhantes de terceira geração produzidos por outros fabricantes de computador realmente satisfizeram a maioria de seus clientes razoavelmente bem. Eles também popularizaram várias técnicas-chave ausentes em sistemas operacionais de segunda geração. Provavelmente a mais importante destas era a **multiprogramação.** No 7094, quando o *job* atual fazia uma pausa para esperar uma fita ou outra operação de E/S completar-se, a CPU simplesmente ficava desocupada até o término da operação. Com cálculos científicos que exigem intensamente a CPU, as operações de E/S são pouco freqüentes, de modo que esse tempo desperdiçado não é significativo. Com processamento comercial de dados, o tempo de espera de E/S freqüentemente pode ser 80 ou 90% do tempo total. Então, algo devia ser feito para evitar ter a CPU desocupada durante tanto tempo.

A solução desenvolvida foi dividir a memória em várias partições, com um *job* diferente em cada partição, como mostrado na Figura 1-4. Enquanto um *job* estava esperando uma operação de E/S completar-se, outro *job* podia usar a CPU. Se um número suficiente de *jobs* pudesse ser mantido na memória principal de uma vez, a CPU poderia ficar ocupada quase 100% do tempo. Ter múltiplos *jobs* em memória de uma vez requer hardware especial para evitar que um *job* prejudique outro, mas o 360 e outros sistemas de terceira geração estavam equipados com esse hardware.

Outro importante recurso apresentado nos sistemas operacionais de terceira geração era a capacidade de ler *jobs* de cartões para o disco logo que eles eram trazidos para a sala de computador. Então, sempre que um *job* em execução acabava, o sistema operacional podia carregar um novo *job* do disco na partição agora vazia e executá-lo. Essa técnica é chamada *spooling* (de *Simultaneous Peripheral Operation On Line* — Operação Periférica Simultânea *On Line*) e era também utilizada para saída. Com o *spooling*, os 1401 não eram mais necessários e acabava grande parte do trabalho de carregamento de fitas.

Embora os sistemas operacionais de terceira geração servissem bem para grandes cálculos científicos e para aplicações comerciais com grande volume de processamento de dados, eles eram ainda basicamente sistemas de lote. Muitos programadores sentiam falta das máquinas de primeira geração quando tinham a máquina toda para si por algumas horas e, assim, podiam depurar seus programas rapidamente. Com os sistemas de terceira geração, o tempo entre submeter um *job* e receber a saída era freqüentemente de várias horas, então uma única vírgula malcolocada podia causar uma falha na compilação e o programador perdia metade de um dia.

Essa necessidade de tempo de resposta rápido preparou o caminho para o **compartilhamento de tempo,** uma variante da multiprogramação na qual cada usuário tinha um terminal *on-line*. Em um sistema de tempo com-

*N. de T. *Bug:* um erro na codificação ou na lógica que faz com que um programa não funcione corretamente ou que produza resultados incorretos.

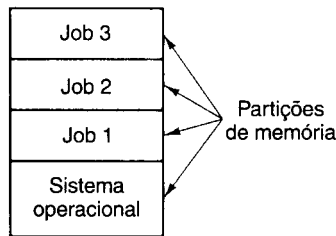



Figura 1-4 Um sistema de multiprogramação com três *jobs* em memória.

partilhado, se 20 usuários efetuavam *logon** e 17 deles estavam pensando, conversando ou bebendo café, a CPU podia estar alocada para os três *jobs* que requeriam processamento. Como as pessoas que depuram programas normalmente utilizam comandos curtos (p. ex., compilar um procedimento de cinco páginas) em vez de longos (p. ex., classificar um arquivo de um milhão de registros), o computador pode oferecer serviço rápido e interativo para alguns usuários e talvez também trabalhar com grandes trabalhos em lote em segundo plano quando a CPU está desocupada. Embora o primeiro sistema de tempo compartilhado sério (CTSS) tenha sido desenvolvido no M.I.T. em um 7094 especialmente modificado (Corbato *et al.*, 1962), o compartilhamento de tempo não se popularizou realmente até que a necessária proteção de hardware se tornasse comum durante a terceira geração.

Depois do êxito do sistema CTSS, o MIT, a Bell Labs e a General Electric (na época, um importante fabricante de computadores) decidiram dedicar-se ao desenvolvimento de um "*computer utility*"**, uma máquina que suportaria centenas de usuários de tempo compartilhado simultâneos. Seu modelo era o sistema de distribuição de eletricidade — quando você precisa de energia elétrica, basta simplesmente introduzir um plugue na tomada da parede e, dentro do possível, toda a energia de que você precisa estará disponível. Os projetistas desse sistema, conhecido como MULTICS (*MULTiplexed Information and Computing Service* — Serviço de Computação e Informação Multiplexado), vislumbraram uma máquina para oferecer enorme poder de cálculo a todos em Boston. A idéia de que máquinas muito mais poderosas do que seu GE-645 seriam vendidas como computadores pessoais por alguns milhares de dólares só 30 anos mais tarde era pura ficção científica na época.

Para encurtar essa longa história, o MULTICS introduziu muitas idéias seminais na literatura sobre computadores, mas construí-lo foi algo muito mais difícil do que o esperado. A Bell Labs retirou-se do projeto e a General Electric desistiu completamente do negócio de computadores. Por fim, o MULTICS rodou suficientemente bem para ser utilizado em um ambiente de produção no MIT e dúzias de outros lugares, mas o conceito de um *computer utility* virou um fiasco à medida que os preços dos computadores despencaram. Ainda assim, o MULTICS teve uma influência enorme em sistemas subseqüentes. Ele é descrito em Corbato e colaboradores, 1972, Corbato e Vyssotsky, 1965, Daley e Dennis, 1968, Organick, 1972, e Saltzer, 1974.

Outro importante desenvolvimento durante a terceira geração foi o crescimento fenomenal dos minicomputadores, começando com o PDP-1 da DEC em 1961. O PDP-1 tinha só 4K de palavras de 18 bits, mas a US$ 120.000 por máquina (menos que 5% do preço de um 7094) venderam como pãozinho quente de padaria. Para certas espécies de trabalhos não-numéricos, era quase tão rápido quanto os 7094 e deu origem a toda uma nova indústria, tendo sido rapidamente seguido por uma série de outros PDP (mas, ao contrário da família IBM, eram todos incompatíveis), culminando no PDP-11.

Um dos cientistas de computação da Bell Labs que tinha trabalhado no projeto do MULTICS, Ken Thompson, subseqüentemente descobriu um pequeno microcomputador PDP-7 que ninguém estava usando e começou a escrever uma versão simplificada monousuária do MULTICS. Esse trabalho mais tarde desenvolveu-se no sistema operacional UNIX®, que se tornou popular no mundo acadêmico, entre órgãos do governo e entre muitas empresas.

A história do UNIX foi bem contada em outros textos (p. ex., Salus, 1994). Basta dizer que como o código-fonte estava amplamente disponível, várias organizações desenvolveram suas próprias versões (incompatíveis), que levaram ao caos. Para tornar possível escrever programas que podiam executar em qualquer sistema UNIX, o IEEE desenvolveu um padrão para o UNIX, chamado **POSIX**, que a maioria das versões do UNIX agora suporta. O POSIX define uma interface mínima de chamadas de sistema que sistemas compatíveis com o UNIX devem suportar. Aliás, alguns outros sistemas operacionais agora também suportam a interface POSIX.

*N. de R. *Logon*: Processo de identificação do usuário para o computador, depois que entra em contato com ele através de uma linha de comunicação. Também chamado de *login* (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998).

**N. de R. Utility neste caso tem o sentido de um serviço público, indicando um recurso computacional amplamente disponível.

1.2.4 A Quarta Geração (1980-hoje): Computadores Pessoais

O desenvolvimento dos circuitos LSI (*Large Scale Integration* – integração em larga escala), chips contendo milhares de transistores por centímetro quadrado de silício, foi o alvorecer da era do computador pessoal. Em termos de arquitetura, tais computadores não eram tão diferentes dos minicomputadores da classe PDP-11, mas, em termos de preço, certamente eram diferentes. Assim como o minicomputador possibilitou que um departamento em uma empresa ou em uma universidade tivesse seu próprio computador, o microprocessador tornou possível que uma pessoa tivesse seu próprio computador pessoal. Os computadores pessoais mais poderosos utilizados em negócios, em universidades e em órgãos do governo normalmente são chamados de **estações de trabalho,** mas na realidade são apenas computadores pessoais de maior porte, normalmente conectados por uma rede.

A ampla disponibilidade do poder de computação, especialmente a computação altamente interativa, normalmente com excelentes gráficos, levou ao crescimento de uma importante indústria de produção de software para computadores pessoais. Grande parte desse software tinha uma **interface amigável**, ou seja, projetada para usuários que não só ignoravam tudo sobre computadores como também não tinham nenhuma intenção de aprender. Essa era certamente uma mudança importante do OS/360, cuja linguagem de controle, JCL (*Job Control Language* – linguagem de controle de trabalhos), era tão complexa que livros inteiros foram escritos sobre ela (p. ex., Cadow, 1970).

Dois sistemas operacionais inicialmente dominaram a cena do computador pessoal e da estação de trabalho: o MS-DOS da Microsoft e o UNIX. O MS-DOS era amplamente utilizado no IBM PC e em outras máquinas que empregam a CPU Intel 8088 e seus sucessores, 80286, 80386 e 80486 (que referiremos doravante como 286, 386 e 486, respectivamente) e mais tarde o Pentium e o Pentium PRO. Embora a versão inicial do MS-DOS fosse relativamente primitiva, as posteriores incluíram recursos mais avançados, inclusive muitos tomados do UNIX. O sucessor da Microsoft para o MS-DOS, o WINDOWS, originalmente rodava por cima do MS-DOS (i. e., era mais como um *shell* do que um sistema operacional verdadeiro), mas, em 1995, foi lançada uma versão "independente" do WINDOWS, o WINDOWS 95®, de modo que o MS-DOS não é mais necessário para suportá-lo. Outro sistema operacional da Microsoft é o WINDOWS NT, compatível com o WINDOWS 95 até certo nível, mas que foi completamente reescrito internamente.

O outro importante concorrente é o UNIX, que é dominante em estações de trabalho e em outros computadores topo de linha, como servidores de rede, e especialmente popular em máquinas equipadas com chips RISC de alto desempenho. Essas máquinas normalmente têm o poder de computação de um minicomputador, ainda que dedicados para um único usuário, então é lógico que sejam equipadas com um sistema operacional originalmente projetado para minicomputadores, o UNIX.

Um desenvolvimento interessante que começou durante meados da década de 80 é o crescimento de redes de computadores pessoais executando **sistemas operacionais de rede** e **sistemas operacionais distribuídos** (Tanenbaum, 1995). Em um sistema operacional de rede, os usuários estão cientes da existência de múltiplos computadores e podem conectar-se a máquinas remotas e copiar arquivos de uma máquina para outra. Cada máquina executa localmente o sistema operacional e tem seu próprio usuário local (ou usuários).

Os sistemas operacionais de rede não são fundamentalmente diferentes dos sistemas operacionais monoprocessador. Eles obviamente necessitam de uma interface de rede e de algum software de baixo nível para controlá-la, assim como programas para permitir conexões remotas e acesso a arquivos remotos. Entretanto, tais adições não mudam essencialmente a estrutura do sistema operacional.

Um sistema operacional distribuído, ao contrário, é aquele que aparece para seus usuários como um sistema monoprocessado tradicional, mesmo que realmente seja composto de múltiplos processadores. Os usuários não sabem onde seus programas estão sendo executados nem onde seus arquivos estão localizados; tudo isso deve ser manipulado de forma automática e eficiente pelo sistema operacional.

Verdadeiros sistemas operacionais distribuídos requerem mais do que apenas adicionar uma pequena porção de código a um sistema operacional monoprocessado, porque os sistemas centralizados e distribuídos diferem significativamente. Os sistemas distribuídos, por exemplo, com freqüência permitem que os aplicativos executem em vários processadores ao mesmo tempo; assim eles requerem um algoritmo de agendamento de processador mais complexo a fim de otimizar a quantidade de paralelismo.

A demora de comunicação dentro de uma rede seguidamente significa que esses (e outros) algoritmos devem executar com informações desatualizadas, incompletas ou até incorretas. Essa situação é radicalmente diferente de um sistema monoprocessado em que o sistema operacional tem informações completas sobre o estado do sistema.

1.2.5 A História do MINIX

Quando o UNIX era jovem (Versão 6), o código-fonte estava amplamente disponível, sob licença da AT&T, e era muito estudado. John Lions, da Universidade New South Wales, na Austrália, escreveu uma pequena brochura que descrevia sua operação, linha por linha (Lions, 1996). Essa publicação era utilizada (com permissão de AT&T) como referência em muitos cursos universitários sobre sistemas operacionais.

Quando a AT&T lançou a Versão 7, começou-se a perceber que o UNIX era um produto comercial valioso, e assim ela lançou essa versão com uma licença proibindo que

o código-fonte fosse estudado em cursos, para evitar pôr em risco seu status de segredo de negócio. Muitas universidades tiveram de conformar-se em simplesmente acabar com o estudo de UNIX e ensinar só teoria.

Infelizmente, ensinar só teoria deixa o aluno com uma visão equivocada do que é realmente um sistema operacional. Os temas teóricos que normalmente são abordados detalhadamente em cursos e em livros sobre sistemas operacionais, como algoritmos de agendamento, não são realmente tão importantes na prática. Os assuntos realmente relevantes, como E/S e sistemas de arquivos, geralmente são negligenciados, pois há pouca teoria sobre eles.

Para corrigir essa situação, um dos autores deste livro (Tanenbaum) decidiu escrever um novo sistema operacional a partir do zero, que seria compatível com UNIX do ponto de vista do usuário, mas completamente diferente interiormente. Por não usar sequer uma linha do código da AT&T, esse sistema evita as restrições de licenciamento, assim ele pode ser utilizado para estudo individual ou em classe. Desta maneira, os leitores podem dissecar um sistema operacional real para ver o que há por dentro, assim como estudantes de Biologia dissecam rãs em laboratório. O nome MINIX significa mini-UNIX pois ele é tão pequeno que mesmo um não-especialista pode entender seu funcionamento.

Além da vantagem de eliminar os problemas legais, o MINIX tem outra vantagem sobre o UNIX. Foi escrito uma década depois do UNIX e estruturado de maneira mais modular. O sistema de arquivos do MINIX, por exemplo, não é absolutamente parte do sistema operacional, mas roda como um programa de usuário. Outra diferença: enquanto o UNIX foi projetado para ser eficiente, o MINIX o foi para ser legível (se é que alguém pode falar de qualquer programa com centenas de páginas como sendo legível). O código do MINIX, por exemplo, tem milhares de comentários.

O MINIX originalmente foi projetado para ter compatibilidade com a Versão 7 (V7) do UNIX, a qual era utilizada como modelo por causa de sua simplicidade e elegância. Às vezes, diz-se que a Versão 7 não era só uma melhora em relação a todos os seus antecessores, como também sobre todos seus sucessores. Com o advento do POSIX, o MINIX começou a desenvolver-se em direção ao novo padrão, mas ainda mantendo retrocompatibilidade com programas existentes. Essa espécie de evolução é comum na indústria dos computadores, na medida em que nenhum fabricante iria querer lançar um sistema que nenhum de seus clientes pudesse utilizar sem passar por grandes adaptações. A versão do MINIX, descrita neste livro, é baseada no padrão do POSIX (diferente da versão descrita na primeira edição, que era baseada na V7).

Como o UNIX, o MINIX foi escrito na linguagem de programação C e projetado para ser facilmente portado para vários computadores. A implementação inicial era para o IBM PC, pois esse computador é mais amplamente utilizado. Posteriormente, ele foi portado para computadores Atari, Amiga, Macintosh e SPARC. Para manter-se fiel à filosofia "quanto menor, melhor", o MINIX originalmente não exigia sequer um disco rígido, trazendo-o assim para alcance do orçamento de muitos alunos (por mais que pareça surpreendente hoje, em meados da década de 80 quando o MINIX nascia, os discos rígidos ainda eram uma cara novidade). À medida que o MINIX crescia em funcionalidade e tamanho, acabou chegando um momento em que um disco rígido era necessário, mas fiel à filosofia do MINIX, uma partição de 30 megabytes é suficiente. Em contraste, alguns sistemas comerciais UNIX agora recomendam, pelo menos, uma partição de disco de 200MB como o mínimo.

Para o usuário médio que utiliza um IBM PC, rodar o MINIX é semelhante a rodar o UNIX. Muitos dos programas básicos, como *cat*, *grep*, *ls*, *make*, e o *shell* estão presentes e executam as mesmas funções que seus correspondentes no UNIX. Como o sistema operacional em si, todos esses programas utilitários foram reescritos completamente a partir do zero pelo autor e por seus alunos entre outras pessoas dedicadas.

Ao longo deste livro, o MINIX será utilizado como um exemplo. Mas a maioria dos comentários sobre o MINIX, exceto aqueles sobre o código em si, também aplica-se ao UNIX. Muitos deles também aplicam-se a outros sistemas. Essa observação deve ser mantida em mente durante a leitura do texto.

Como um parêntese, algumas palavras sobre o LINUX e seu relacionamento com o MINIX podem ser de interesse para alguns leitores. Logo depois que o MINIX foi lançado, um grupo de discussão da USENET foi criado para discuti-lo. Em algumas semanas o grupo tinha 40.000 assinantes, grande parte dos quais queria adicionar um grande número de novos recursos ao MINIX para torná-lo maior e melhor (bem, pelo menos maior). Todos os dias, várias centenas deles ofereciam sugestões, idéias e pequenos trechos de código. O autor do MINIX resistiu com êxito a esse assalto por vários anos para manter o MINIX suficientemente pequeno e limpo para os alunos entenderem-no. Gradualmente, começou a tornar-se evidente o que ele realmente significava. Nesse ponto, um estudante finlandês, Linus Torvalds, decidiu escrever um clone do MINIX projetado para ser um sistema de produção carregado de recursos, em vez de uma ferramenta educacional. Assim nascia o LINUX.

1.3 CONCEITOS DE SISTEMA OPERACIONAL

A interface entre o sistema operacional e os programas de usuário é definida pelo conjunto de "instruções estendidas" que o sistema operacional proporciona. Essas instruções estendidas tradicionalmente foram conhecidas como **chamadas de sistema** (*system calls*), embora possam ser implementadas de várias maneiras hoje. Para realmente entender o que os sistemas operacionais fazem, devemos examinar essa interface bem de perto. As chamadas dispo-

níveis na interface variam de sistema operacional para sistema operacional (embora os conceitos subjacentes tendam a ser semelhantes).

Assim, somos forçados a fazer uma escolha entre (1) generalidades vagas ("sistemas operacionais têm chamadas de sistema para ler arquivos") e (2) algum sistema específico ("o MINIX tem uma chamada de sistema READ com três parâmetros: um para especificar o arquivo, um para dizer onde os dados devem ser colocados e um para indicar quantos bytes serão lidos").

Escolhemos a última abordagem. Ela é mais trabalhosa, mas oferece uma visão melhor sobre o que os sistemas operacionais realmente fazem. Na Seção 1.4, veremos mais de perto as chamadas de sistema presentes tanto no UNIX como no MINIX. Para simplificar, iremos referir-nos só ao MINIX, mas as chamadas de sistema correspondentes do UNIX são baseadas no POSIX na maioria dos casos. Antes de examinarmos as chamadas de sistema reais, porém, vale a pena dar uma rápida olhada no MINIX, para obter uma visão geral do que é um sistema operacional como um todo. Essa visão geral aplica-se igualmente bem ao UNIX.

As chamadas de sistema do MINIX dividem-se grosseiramente em duas categorias amplas: aquelas que lidam com processos e aquelas que lidam com o sistema de arquivos. Examinaremos agora cada uma delas separadamente.

1.3.1 Processos

Um conceito-chave no MINIX e em todos sistemas operacionais é o **processo.** Um processo é basicamente um programa em execução. Associado com cada processo está seu **espaço de endereçamento,** uma lista de locais da memória a partir de um mínimo (normalmente 0) até um máximo, que o processo pode ler e gravar. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Também associado com cada processo está um conjunto de registradores, incluindo o contador de programa, o ponteiro da pilha e outros registradores de hardware e todas as demais informações necessárias para executar o programa.

Voltaremos ao conceito de processo em muito mais detalhe no Capítulo 2, mas, por enquanto, a maneira fácil de obter uma boa noção de um processo é pensar nos sistemas de tempo compartilhado. Periodicamente, o sistema operacional decide parar de executar um processo e começar a executar outro, por exemplo, porque o primeiro teve mais que sua parte de tempo de CPU no segundo passado.

Quando um processo é suspenso temporariamente como esse, mais tarde deve ser reiniciado exatamente no mesmo estado em que estava quando foi suspenso. Isso significa que todas as informações sobre o processo devem ser explicitamente salvas em algum lugar durante a suspensão. Por exemplo, o processo pode ter vários arquivos abertos para leitura. Associado a cada um desses arquivos está um ponteiro que informa a posição atual (i. e., o número do byte ou registro a ser lido em seguida). Quando um processo é suspenso temporariamente, todos esses ponteiros devem ser salvos para que uma chamada READ executada depois que o processo é reiniciado leia os dados adequados. Em muitos sistemas operacionais, todas as informações sobre cada processo, que não o conteúdo do seu próprio espaço de endereçamento, são armazenadas em uma tabela do sistema operacional chamada **tabela de processos**, que é uma matriz (ou lista encadeada) de estruturas, uma para cada processo atualmente existente.

Assim, um processo (suspenso) consiste em seu espaço de endereço, normalmente chamado **imagem de núcleo** (em homenagem às memórias de núcleo magnético utilizadas antigamente), e sua entrada na tabela de processos, que contém seus registradores, entre outras coisas.

As chamadas-chave do sistema de gerenciamento de processos são as que lidam com a criação e com o encerramento de processos. Considere um exemplo típico. Um processo chamado **interpretador de comandos** ou ***shell*** lê comandos de um terminal. O usuário digitou um comando que requisita a compilação de um programa. O *shell* agora deve criar um novo processo que executará o compilador. Quando esse processo termina a compilação, ele executa uma chamada de sistema para encerrar-se.

Se um processo pode criar um ou mais processos (referidos como **processos-filho**), os quais podem criar processos-filho, rapidamente chegamos à estrutura de árvore de processos da Figura 1-5. Processos relacionados que estão cooperando para executar uma tarefa freqüentemente precisam comunicar-se entre si e sincronizar suas atividades. Essa comunicação é chamada de **comunicação interprocessos** e será abordada detalhadamente no Capítulo 2.

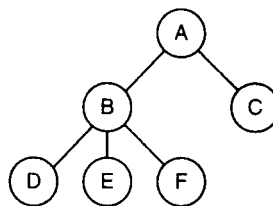



Figura 1-5 Uma árvore de processos. O processo A criou dois processos-filho, *B* e *C*. O processo *B* criou três processos-filho, *D*, *E* e *F*.

Outras chamadas de sistema para processos estão disponíveis para requisitar mais memória (ou liberar memória não-utilizada), esperar um processo-filho terminar e substituir seu programa por um diferente.

Ocasionalmente, há necessidade de transmitir as informações para um processo em execução que não está parado aguardando-as. Por exemplo, um processo que está comunicando-se com outro em um computador diferente faz isso enviando mensagens por uma rede. Para evitar a possibilidade de que uma mensagem ou sua resposta seja perdida, o remetente pode requisitar que o próprio sistema operacional notifique-o depois de um número especificado de segundos, de modo que ele possa retransmitir a mensagem se nenhuma confirmação foi recebida ainda. Depois de configurar esse temporizador, o programa pode continuar fazendo outro trabalho.

Quando o número especificado de segundos passou, o sistema operacional envia um **sinal** para o processo. O sinal faz com que o processo suspenda temporariamente o que estava fazendo, salve seus registradores na pilha e comece a executar um procedimento especial de tratamento de sinal, por exemplo, para retransmitir uma mensagem presumivelmente perdida. Quando a manipulação do sinal estiver concluída, o processo em execução é reiniciado no estado em que estava antes do sinal. Os sinais são o análogo de software das interrupções de hardware e podem ser gerados por uma variedade de causas além da expiração de temporizadores. Muitas exceções detectadas por hardware, como executar uma instrução ilegal ou usar um endereço inválido, também são convertidas em sinais para o processo causador.

A cada pessoa autorizada a utilizar o MINIX é designado com uma ***uid*** (de *user identification* — identificação de usuário) pelo administrador de sistema. Cada processo iniciado no MINIX tem a *uid* da pessoa que o iniciou. Um processo-filho tem a mesma *uid* que seu pai. Uma *uid*, denominada **superusuário,** tem poder especial e pode transgredir muitas das regras de proteção. Em instalações grandes, só o administrador de sistema sabe a senha necessária para tornar-se superusuário, mas muitos dos usuários comuns (especialmente estudantes) despendem esforço considerável tentando localizar defeitos no sistema que os permitam tornar-se superusuário sem a senha.

1.3.2 Arquivos

A outra categoria ampla de chamadas de sistema está relacionada com o sistema de arquivos. Como observado anteriormente, uma função importante do sistema operacional é esconder as peculiaridades dos discos e outros dispositivos de E/S e apresentar ao programador um amigável e limpo modelo abstrato de arquivos independentes de dispositivo. As chamadas de sistema obviamente são necessárias para criar, para remover, para ler e para escrever arquivos. Antes de um arquivo poder ser lido, ele deve ser aberto e depois de lido deve ser fechado, assim chamadas são fornecidas para fazer tais coisas.

Para fornecer um lugar para manter os arquivos, o MINIX tem o conceito de **diretório** como uma maneira de agrupar os arquivos. Um aluno, por exemplo, talvez tenha um diretório para cada curso que ele está fazendo (para os programas necessários para esse curso), outro diretório para seu correio eletrônico e ainda um terceiro diretório para sua *home page* na World Wide Web. As chamadas de sistema, então, são necessárias para criar e para remover diretórios. Chamadas também são fornecidas para inserir um arquivo em um diretório existente, bem como para remover um arquivo de um diretório. As entradas de diretório podem ser arquivos ou outros diretórios. Esse modelo também origina uma hierarquia — o sistema de arquivos —, como mostrado na Figura 1-6.

Tanto as hierarquias de processos quanto as de arquivos são organizadas como árvores, mas as semelhanças param por aí. As hierarquias de processos normalmente não são muito profundas (mais de três níveis é raro), ao passo que as hierarquias de arquivo têm comumente quatro, cinco e até mais níveis de profundidade. Além disso, as hierarquias de processos normalmente têm vida curta, em geral, alguns minutos no máximo, enquanto as hierarquias de diretórios podem existir durante anos. A posse e a proteção também diferem para processos e para arquivos. Normalmente, só um processo-pai pode controlar ou mesmo acessar um processo-filho, mas quase sempre existem mecanismos para permitir que arquivos e diretórios sejam lidos por um grupo mais amplo que apenas o proprietário.

Cada arquivo dentro da hierarquia de diretórios pode ser especificado dando-se seu **nome de caminho** a partir do topo da hierarquia de diretórios, o **diretório-raiz.** Esses nomes de caminho absolutos consistem na lista de diretórios que deve ser percorrida a partir do diretório-raiz para alcançar o arquivo, com barras separando os componentes. Na Figura 1-6, o caminho para o arquivo *CS101* é */Faculdade/Prof.Brown/Cursos/CS101.* A primeira barra indica que o caminho é absoluto, isto é, inicia-se no diretório-raiz.

A cada instante, cada processo tem um **diretório de trabalho** atual, onde os nomes de caminho que não começam com uma barra são procurados. Como um exemplo, na Figura 1-6, se */Faculdade/Prof.Brown* fosse o diretório de trabalho, então, o uso do nome de caminho *Cursos/CS101* resultaria no mesmo arquivo que o nome de caminho absoluto dado acima. Os processos podem mudar seu diretório de trabalho emitindo uma chamada de sistema que especifica o novo diretório de trabalho.

Os arquivos e os diretórios no MINIX são protegidos designando um código binário de proteção de 9 bits a cada um. O código de proteção consiste em três campos de 3 bits, um para o proprietário, um para outros membros do grupo do proprietário (os usuários são divididos em grupos pelo administrador do sistema) e um terceiro para todas as demais pessoas. Cada campo tem um bit para acesso de leitura (*read*), um bit para acesso de gravação (*write*) e um bit para acesso de execução. Esses 3 bits são conhecidos como **bits rwx.** Por exemplo, o código de proteção

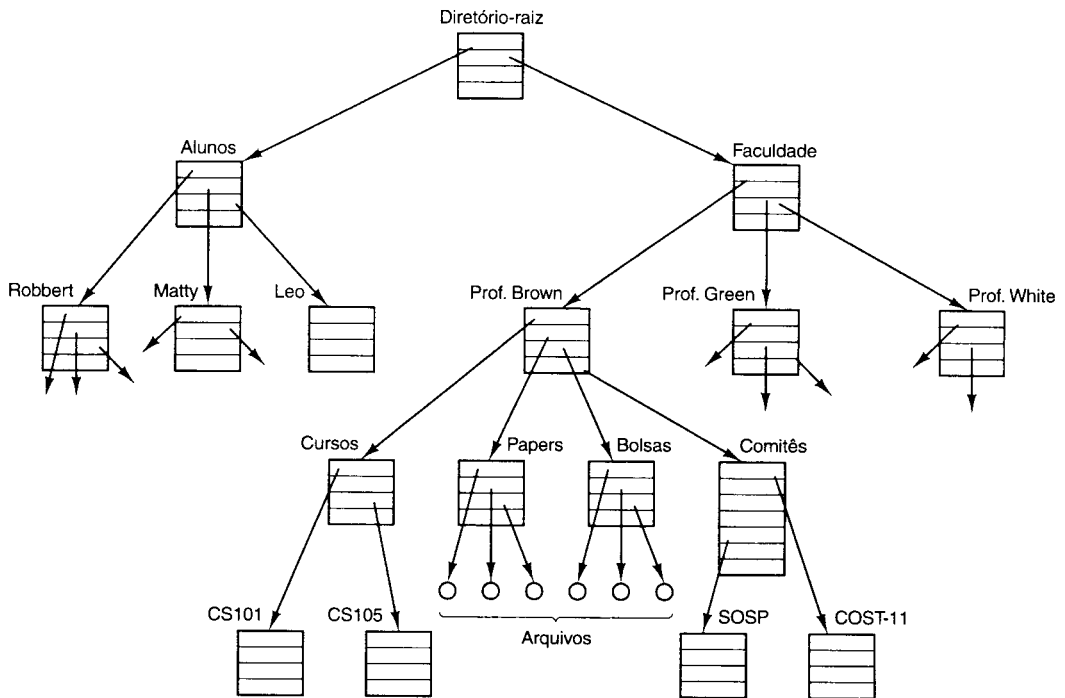



Figura 1-6 Um sistema de arquivos para um departamento de uma universidade.

rwxr-x-x significa que o proprietário pode ler, gravar ou executar o arquivo, outros membros de grupo podem ler ou executar (mas não gravar) o arquivo e todas as demais pessoas podem executar (mas não ler ou gravar) o arquivo. Para um diretório, x indica permissão de pesquisa. Um traço significa que a permissão correspondente está ausente.

Antes de um arquivo poder ser lido ou gravado, ele deve ser aberto, momento em que as permissões são verificadas. Se o acesso for permitido, o sistema retorna um número inteiro pequeno chamado **descritor de arquivo** para usar em operações subseqüentes. Se o acesso for proibido, um código de erro é retornado.

Outro conceito importante no MINIX é o sistema de arquivos montados. Quase todos os computadores pessoais têm uma ou mais unidades de disquete onde os disquetes podem ser inseridos e removidos. Para fornecer uma maneira limpa de lidar com essas mídias removíveis (e também CD-ROMs, que são também removíveis), o MINIX permite que o sistema de arquivos em um disquete seja unido à árvore principal. Considere a situação da Figura 1-7(a). Antes da chamada MOUNT, o disco de RAM (disco simulado na memória principal) contém o **sistema de arquivos-raiz** ou **primário,** e a unidade 0 contém um disquete que, por sua vez, contém outro sistema de arquivos.

Entretanto, o sistema de arquivos na unidade 0 não pode ser utilizado porque não há como especificar nomes de caminho nele. O MINIX não permite que os nomes de caminho tenham como prefixo um nome de unidade ou um número; isso seria precisamente o tipo de dependência de dispositivo que os sistemas operacionais deveriam eliminar. Em vez disso, a chamada de sistema MOUNT permite que o sistema de arquivos na unidade 0 seja anexado ao sistema de arquivos raiz onde quer que o programa queira que ele esteja. Na Figura 1-7(b), o sistema de arquivos na unidade 0 foi montado no diretório *b*, permitindo, assim, acesso aos arquivos */b/x* e */b/y*. Se o diretório *b* contivesse qualquer arquivo, esse não seria acessível enquanto a unidade 0 estivesse montada, uma vez que */b* iria referir-se ao diretório-raiz da unidade 0. (Ser incapaz de acessar esses arquivos não é tão sério quanto a princípio pode parecer: os sistemas de arquivos quase sempre são montados em diretórios vazios.)

Outro conceito importante no MINIX é o de **arquivo especial**, fornecido para fazer dispositivos de E/S parecerem arquivos. Assim, eles podem ser lidos e gravados usando as mesmas chamadas de sistema utilizadas para ler e para gravar arquivos. Há dois tipos de arquivos especiais: **arquivos especiais de bloco** e **arquivos especiais de caractere.** Os arquivos especiais de bloco são utilizados

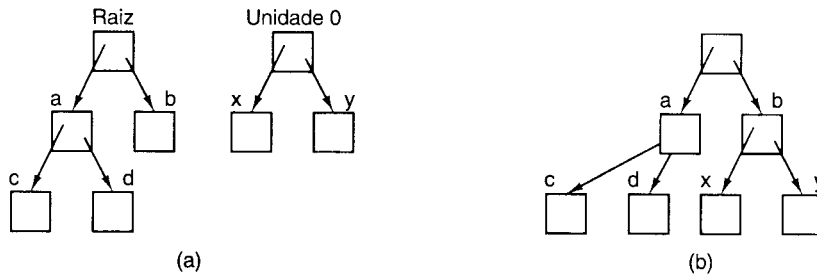



Figura 1-7 (a) Antes da montagem, os arquivos na unidade 0 não são acessíveis. (b) Depois da montagem, eles são parte da hierarquia de arquivos.

para modelar dispositivos que consistem em uma coleção de blocos aleatoriamente endereçáveis, como discos. Abrindo um arquivo especial de bloco e lendo, digamos, o bloco 4, um programa pode acessar diretamente o quarto bloco no dispositivo, sem considerar a estrutura do sistema de arquivos contida nele. De maneira semelhante, arquivos especiais de caractere são utilizados para modelar impressoras, modems e outros dispositivos que aceitam ou dão como saída um fluxo de caracteres.

O último recurso que discutiremos nesta visão geral está relacionado tanto com processos como com arquivos: *pipes*. Um ***pipe*** é um tipo de pseudo-arquivo que pode ser utilizado para conectar dois processos (Figura 1-8 adiante). Quando o processo *A* quer enviar dados para o processo *B*, ele grava no *pipe* como se fosse um arquivo de saída. O processo *B* pode ler os dados lendo a partir do *pipe* como se fossem um arquivo de entrada. Assim, a comunicação entre processos no MINIX parece-se muito com a leitura e com a gravação comuns de arquivos. Além de tudo, a única maneira como um processo pode descobrir que o arquivo de saída que ele está gravando não é realmente um arquivo, mas um *pipe*, é fazendo uma chamada especial de sistema.

1.3.3 O *Shell*

O sistema operacional MINIX é o código que executa as chamadas de sistema. Os editores, compiladores, *assemblers*, montadores e interpretadores de comando definitivamente não são parte do sistema operacional, ainda que sejam importantes e úteis. Sob o risco de confundir um pouco as coisas, veremos nesta seção resumidamente, o interpretador de comandos do MINIX, chamado ***shell***, que, embora não seja parte do sistema operacional, faz uso pesado de muitos recursos do sistema operacional e, assim, serve como um bom exemplo de como as chamadas de sistema podem ser utilizadas. Ele também é a interface primária entre o usuário à frente do terminal e o sistema operacional.

Quando qualquer usuário efetua *logon*, um *shell* é inicializado. O *shell* tem o terminal como entrada-padrão e saída-padrão. Ele inicia apresentando o ***prompt***, um caractere como o sinal de cifrão, que informa ao usuário que o *shell* está esperando receber um comando. Se o usuário agora digitar

```
date
```

por exemplo, o *shell* cria um processo-filho e executa o programa *date* como o filho. Enquanto o processo-filho está executando, o *shell* espera que ele termine. Quando o filho termina, o *shell* exibe o *prompt* novamente e tenta ler a próxima linha de entrada.

O usuário pode especificar que a saída-padrão seja redirecionada para um arquivo, por exemplo,

```
date >file
```

De maneira semelhante, a entrada-padrão pode ser redirecionada, como em

```
sort <file1 >file2
```

que invoca o programa *sort* que pega a entrada de *file1* e a envia ao arquivo de saída *file2*.

A saída de um programa pode ser utilizada como a entrada para outro programa, conectando-os com um *pipe*. Assim,

```
cat file1 file2 file3 | sort >/dev/lp
```

invoca o programa *cat* para concatenar três arquivos e enviar a saída para *sort* organizar todas as linhas em ordem alfabética. A saída de *sort* é redirecionada para o arquivo */dev/lp*, que é um típico nome de arquivo especial de caractere para impressora. (Por convenção, todos os arquivos especiais são mantidos no diretório */dev*.)

Se um usuário colocar o caractere & depois de um comando, o *shell* não o espera completar. Em vez disso, ele simplesmente apresenta o *prompt* imediatamente. Conseqüentemente,

```
cat file1 file2 file3 | sort >/dev/lp &
```

inicializa *sort* como um *job* em segundo plano, permitindo que o usuário continue a trabalhar normalmente en-

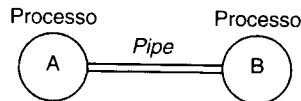



Figura 1-8 Dois processos conectados por um *pipe*.

quanto *sort* está em execução. O *shell* tem diversos outros recursos interessantes que não temos espaço para discutir aqui. Consulte qualquer uma das referências sugeridas sobre o UNIX para obter mais informações sobre o *shell*.

1.4 CHAMADAS DE SISTEMA

Munidos de nosso conhecimento geral sobre como o MINIX lida com processos e arquivos, podemos começar, agora, a examinar a interface entre o sistema operacional e seus programas aplicativos, isto é, o conjunto de chamadas de sistema. Embora essa discussão refira-se especificamente ao POSIX (International Standard 9945-1), e portanto também ao MINIX, a maioria dos outros sistemas operacionais modernos têm chamadas de sistema que executam as mesmas funções, ainda que alguns detalhes possam ser diferentes. Como a verdadeira mecânica de uma chamada de sistema depende muito da máquina e freqüentemente deve ser expressa em código *assembler*, uma biblioteca de procedimentos é fornecida para tornar possível fazer chamadas de sistema a partir de programas em C.

Para tornar o mecanismo de chamadas de sistema mais claro, vamos examinar brevemente um READ. Ele tem três parâmetros: o primeiro especifica o arquivo, o segundo, o buffer e o terceiro especifica o número de bytes a ler. Uma chamada READ de um programa em C pode se parecer com isto:

```
count = read(file, buffer, nbytes);
```

A chamada de sistema (e o procedimento de biblioteca) retorna o número de bytes realmente lido em *count*. Esse valor é normalmente o mesmo que *nbytes*, mas pode ser menor, se, por exemplo, o fim do arquivo for encontrado durante a leitura.

Se a chamada de sistema não puder ser executada, seja devido a um parâmetro inválido, seja devido a um erro de disco, *count* é definido como -1 e o número do erro é colocado em uma variável global, *errno*. Os programas sempre devem verificar os resultados de uma chamada de sistema para ver se um erro ocorreu.

O MINIX tem um total de 53 chamadas de sistema que estão listadas na Figura 1-9, agrupadas por conveniência em seis categorias. Nas seções a seguir, examinaremos resumidamente cada chamada para ver o que elas fazem. De maneira geral, os serviços oferecidos por tais chamadas determinam a maior parte do que o sistema operacional tem de fazer, uma vez que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos em comparação com máquinas de maior porte, com muitos usuários).

Como um parêntese, vale indicar que o significado de chamada de sistema está aberto a interpretações. O padrão POSIX especifica um número de procedimentos que um sistema compatível deve fornecer, mas não se são chamadas de sistema, chamadas de bibliotecas, ou qualquer outra coisa. Em alguns casos, os procedimentos do POSIX são suportados como rotinas de biblioteca em MINIX. Em outros, vários procedimentos necessários são apenas variações menores um do outro, e uma chamada de sistema trata todos eles.

1.4.1 Chamadas de Sistema para Gerenciamento de Processos

O primeiro grupo de chamadas trata do gerenciamento de processos. FORK é um bom lugar para começar a discussão. FORK é o único meio de criar um novo processo. Ele cria uma duplicata exata do processo original, incluindo todos os descritores de arquivos, registradores — tudo. Depois de FORK, o processo original e a cópia (pai e filho) seguem caminhos diferentes. Todas as variáveis têm valores idênticos no momento do FORK, mas como os dados do pai são copiados para criar o filho, posteriores mudanças em um deles não afetam o outro. (O texto, que é imutável, é compartilhado entre pai e filho.) A chamada FORK retorna um valor, que é zero no filho e igual ao identificador de processo (ou **pid**, de **p**rocess **id**entifier) do filho no pai. Usando o *pid* retornado, os dois processos podem ver qual é o processo-pai e qual é o processo-filho.

Na maioria dos casos, depois de um FORK, o filho precisará executar um código diferente do pai. Considere o caso do *shell*. Ele lê um comando do terminal, cria um processo-filho, espera o filho executar o comando e, então, lê o próximo comando quando o filho termina. Para esperar o filho terminar, o pai executa uma chamada de sistema WAITPID, que espera até o filho terminar (qualquer filho, se existir mais de um). WAITPID pode esperar um filho específico ou qualquer filho, definindo o primeiro parâmetro como -1. Quando WAITPID completa-se, o endereço apontado pelo segundo parâmetro contém o status de saída do filho (término normal ou anormal e valor de saída). Várias opções também são oferecidas. A chamada WAITPID substitui a chamada WAIT anterior, que agora é obsoleta, mas continua sendo fornecida por razões de retrocompatibilidade.

Gerenciamento de processos	pid = fork()	Cria um processo-filho idêntico ao pai
	pid = waitpid(pid, &statloc, opts)	Espera um filho terminar
	s = wait(&status)	Versão antiga de waitpid
	s = execve(name, argv, envp)	Substitui a imagem de núcleo de um processo
	exit(status)	Termina a execução do processo e retorna o status
	size = brk(addr)	Define o tamanho do segmento de dados
	pid = getpid()	Retorna o id do processo que fez a chamada
	pid = getpgrp()	Retorna o id do grupo do processo que fez a chamada
	pid = setsid()	Cria uma nova sessão e retorna o id do seu grupo de processo
	l = ptrace(req, pid, addr, data)	Utilizado para depuração
Sinais	s = sigaction(sig, &act, &oldact)	Define a ação a executar em sinais
	s = sigreturn(&context)	Retorna de um sinal
	s = sigprocmask(how, &set, &old)	Examina ou muda a máscara do sinal
	s = sigpending(set)	Obtém o conjunto de sinais bloqueados
	s = sigsuspend(sigmask)	Substitui a máscara de sinal e suspende o processo
	s = kill(pid, sig)	Envia um sinal para um processo
	residual = alarm(seconds)	Define um temporizador
	s = pause()	Suspende o processo que fez a chamada até o próximo sinal
Gerenciamento de arquivos	fd = creat(name, mode)	Modo obsoleto de criar um novo arquivo
	fd = mknod(name, mode, addr)	Cria um nó-i especial, normal, ou de diretório
	fd = open(file, how, ...)	Abre um arquivo para ler, gravar ou ambos
	s = close(fd)	Fecha um arquivo aberto
	n = read(fd, buffer, nbytes)	Lê dados de um arquivo em um buffer
	n = write(fd, buffer, nbytes)	Grava dados de um buffer em um arquivo
	pos = lseek(fd, offset, whence)	Move o ponteiro do arquivo
	s = stat(name, &but)	Obtém informações de status de um arquivo
	s = fstat(fd, &buf)	Obtém informações de status de um arquivo
	fd = dup(fd)	Atribui um novo descritor de arquivo a um arquivo aberto
	s = pipe(&fd[0])	Cria um pipe
	s = ioctl(fd, request, argp)	Executa operações especiais em um arquivo
	s = access(name, amode)	Verifica a acessibilidade do arquivo
	s = rename(old, new)	Atribui um novo nome a um arquivo
	s = fcntl(fd, cmd, ...)	Bloqueio de arquivo e outras operações
Gerenciamento de diretório & de sistema de arquivos	s = mkdir(name, mode)	Cria um novo diretório
	s = rmdir(name)	Remove um diretório vazio
	s = link(namel, name2)	Cria uma nova entrada, name2, apontando para name1
	s = unlink(name)	Remove uma entrada de diretório
	s = mount(special, name, flag)	Monta um sistema de arquivos
	s = umount(special)	Desmonta um sistema de arquivos
	s = sync()	Envia todos os blocos em cache para o disco
	s = chdir(dirname)	Muda o diretório de trabalho
	s = chroot(dirname)	Muda o diretório-raiz
Proteção	s = chmod(name, mode)	Muda os bits de proteção de um arquivo
	uid = getuid()	Obtém o uid do processo que fez a chamada
	gid = getgid()	Obtém o gid do processo que fez a chamada
	s = setuid(uid)	Define o uid do processo que fez a chamada
	s = setgid(gid)	Define o gid do processo que fez a chamada
	s = chown(name, owner, group)	Muda um proprietário do arquivo e grupo
	oldmask = umask(complmode)	Muda a máscara de modo
Gerenciamento de tempo	seconds = time(&seconds)	Obtém o tempo passado desde 1º de Jan., 1970
	s = stime(tp)	Define o tempo passado desde 1º de Jan., 1970
	s = utime(file, timep)	Define o momento de "último acesso" de um arquivo
	s = times(buffer)	Obtém os tempos do usuário e do sistema utilizados até agora

Figura 1-9 As chamadas de sistema do MINIX. O código de retorno é -1 se um erro ocorreu; *fd* é um descritor de arquivo e *n* é contagem de byte. Os outros códigos de retorno são o que o nome sugere.

Agora considere como FORK é utilizado pelo *shell*. Quando um comando é digitado, o *shell* cria um novo processo. Esse processo-filho deve executar o comando do usuário. Ele faz isso usando a chamada de sistema EXEC, que faz com que toda sua imagem de núcleo seja substituída pelo arquivo nomeado em seu primeiro parâmetro. Um *shell* extremamente simplificado ilustrando o uso de FORK, WAITPID e EXEC é mostrado na Figura 1-10.

No caso mais genérico, EXEC tem três parâmetros: o nome do arquivo a ser executado, um ponteiro para a matriz de argumentos e um ponteiro para a matriz de ambiente. Esses serão descritos resumidamente. Várias rotinas de biblioteca, incluindo *execl, execv, execle e execve* são fornecidas para permitir que os parâmetros sejam omitidos ou especificados de diferentes maneiras. Ao longo de todo este livro, usaremos o nome EXEC para representar a chamada de sistema invocada por todas elas.

```
while (TRUE) {                                      /* repete indefinidamente */
  read_command(command, parameters);                /* lê a entrada do terminal */

  if (fork()!= 0) {                                 /* cria o processo-filho */
      /* Código pai. */
      waitpid(-1, &status, 0);                      /* espera o filho encerrar */
  } else {
      /* Código filho. */
      execve(command, parameters, 0);               /* executa o comando */
  }
}
```

Figura 1-10 Um *shell* simplificado. Neste livro, supõe-se que *TRUE* seja definido como 1.

Considere o caso de um comando como

```
cp file1 file2
```

utilizado para copiar *file1* para *file2*. Depois que o *shell* é bifurcado, o processo-filho localiza e executa o arquivo *cp* e passa para ele os nomes dos arquivos de origem e de destino.

O programa principal de *cp* (e o programa principal da maioria dos outros programas) contém a declaração

```
main(argc, argv, envp)
```

onde *o argc é* uma contagem do número de itens na linha de comando, incluindo o nome do programa. Para o exemplo acima, *argc* é 3.

O segundo parâmetro, *argv*, é um ponteiro para uma matriz. O elemento *i* dessa matriz é um ponteiro para a *i*-ésima *string** na linha de comando. Em nosso exemplo, *o argv* [0] apontaria para a *string* "cp". De maneira semelhante, *argv* [1] apontaria para a *string* de 5 caracteres "file1" e *o argv* [2] apontaria para a *string* de 5 caracteres "file2".

O terceiro parâmetro de *main*, *envp*, *é* um ponteiro para o ambiente, uma matriz de *strings* que contém designações na forma *nome* = *valor* utilizados para passar informações como o tipo do terminal e o nome do diretório inicial para um programa. Na Figura 1-10, nenhum ambiente é passado para o filho, então para o terceiro parâmetro de *execve* é zero.

Se EXEC parece complicado, não se desespere; ela é a chamada de sistema mais complexa. Todas as demais são muito mais simples. Como um exemplo simples, considere EXIT, que os processos devem usar quando concluem sua execução. Ela tem um parâmetro, o status de saída (0 a 255), que é retornado para o pai na variável status da chamada de sistema WAIT ou WAITPID. O byte de ordem inferior de status contém o status do término, com 0 sendo término normal e os outros valores sendo várias condições de erro. O byte de ordem superior contém o status de saída do filho (0 a 255). Por exemplo, se um processo-pai executa a instrução

```
n = waitpid(-1, &status, options);
```

ele será suspenso até que algum processo-filho termine. Se o filho sai com, digamos, 4 como parâmetro de *exit*, o pai será despertado com *n* definido como o *pid* do filho e o status definido como 0x0400 (a convenção em C de utilizar o prefixo 0x para constantes hexadecimais será utilizada em todo o livro).

Os processos no MINIX têm sua memória dividida em três segmentos: o **segmento de texto** (i. e., o código do programa), o **segmento de dados** (i. e., as variáveis) e o **segmento de pilha.** O segmento de dados cresce para cima, e a pilha, para baixo, como mostrado na Figura 1-11. Entre eles, há um intervalo de endereços não-utilizados. A pilha cresce automaticamente, conforme o necessário, mas a expansão do segmento de dados é feita explicitamente usando a chamada de sistema BRK. Ela tem um parâmetro, que fornece o endereço onde o segmento de dados deve terminar. Esse endereço pode ser maior que o valor atual (o segmento de dados está aumentando) ou menor que o valor atual (o segmento de dados está diminuindo). O parâmetro deve, naturalmente, ser menor que o ponteiro da pilha; caso contrário, os segmentos de dados e da pilha iriam sobrepor-se, o que é proibido.

Como uma conveniência para o programador, é fornecida uma rotina de biblioteca *sbrk* que também muda o tamanho do segmento de dados, só que seu parâmetro é o número de bytes a adicionar ao segmento de dados (parâmetros negativos reduzem o segmento de dados). Ela funciona acompanhando o tamanho atual do segmento de dados, que é o valor retornado por BRK, calculando o novo tamanho e fazendo uma chamada pedindo esse número de bytes. BRK e SBRK foram considerados demasiadamente dependentes de implementação e não são parte do POSIX.

A próxima chamada de sistema de processo, GETPID, é também a mais simples. Ela apenas retorna o *pid* do processo que fez a chamada. Lembre-se de que em FORK, somente o pai recebeu o *pid* do filho. Se o filho quiser saber o próprio *pid*, deve usar GETPID. A chamada GETPGRP retorna o *pid* do grupo do processo que fez a chamada. SETSID cria uma nova sessão e define o *pid* do grupo como o do proces-

*N. de R. *String*: Cadeia. Uma estrutura de dados composta por uma série de caracteres, geralmente contendo um texto legível e inteligível pelas pessoas. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998).

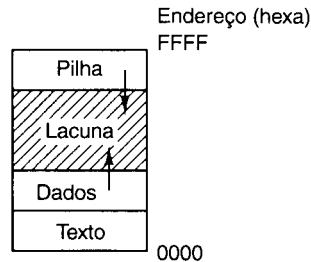



Figura 1-11 Os processos têm três segmentos: texto, dados e pilha. Neste exemplo, os três estão em um espaço de endereço, mas o espaço separado para instruções e para dados também é suportado.

so que fez a chamada. As sessões estão relacionadas a um recurso opcional do POSIX chamado **controle de *job***, que não é suportado pelo MINIX e com o qual não nos ocuparemos mais neste livro.

A última chamada de sistema de gerenciamento de processo, PTRACE, é utilizada para depurar programas controlando o programa que está sendo depurado. Ela permite que o depurador leia e grave a memória do processo controlado e a gerencie de outras maneiras.

1.4.2 Chamadas de Sistema para Sinalização

Embora a maioria das formas de comunicação de interprocessos seja planejada, há situações nas quais uma comunicação inesperada é necessária. Por exemplo, se um usuário acidentalmente instrui um editor de texto a imprimir o conteúdo inteiro de um arquivo muito longo e, então, percebe o erro, é preciso haver alguma maneira de interromper o editor. No MINIX, o usuário pode utilizar a tecla DEL, que envia um sinal para o editor. O editor recebe o sinal e cancela a impressão. Os sinais também podem ser utilizados para informar certas exceções detectadas pelo hardware, como instrução ilegal ou estouro de ponto flutuante. Os limites de tempo (*timeouts*) também são implementados como sinais.

Quando um sinal é enviado para um processo que não anunciou sua receptividade para esse sinal, o processo simplesmente é eliminado sem maior alarde. Para evitar esse destino, um processo pode usar a chamada de sistema SIGACTION para anunciar que está preparado para aceitar algum tipo de sinal e fornecer o endereço do procedimento de manipulação do mesmo e um lugar para armazenar o endereço atual. Depois de uma chamada SIGACTION, se um sinal de tipo relevante (p. ex., a tecla DEL) é gerado, o estado do processo é colocado em sua própria pilha e, então, o manipulador de sinal é chamado. Ele pode executar por quanto tempo quiser e executar qualquer chamada de sistema que necessitar. Na prática, porém, os manipuladores de sinal, em geral, são relativamente curtos. Quando o procedimento de manipulação de sinal está completo, ele chama SIGRETURN para continuar a partir de onde ele saiu antes do sinal. A chamada SIGACTION substitui a antiga chamada SIGNAL, que agora, por razões de retrocompatibilidade é fornecida como um procedimento de biblioteca.

Os sinais podem ser bloqueados no MINIX. Um sinal bloqueado é mantido pendente até que seja desbloqueado. Ele não é enviado, mas também não é perdido. A chamada SIGPROCMASK permite a um processo definir o conjunto de sinais bloqueados apresentando ao *kernel* um mapa de bits. Também é possível para um processo pedir o conjunto de sinais atualmente pendentes, mas que não puderam ser enviados devido ao seu estado bloqueado. A chamada SIGPENDING retorna esse conjunto como um mapa de bits. Finalmente, a chamada SIGSUSPEND permite que um processo defina atomicamente o mapa de bits dos sinais bloqueados e suspenda a si próprio.

Em vez de oferecer uma função para capturar um sinal, o programa também pode especificar a constante SIG_IGN para ignorar todos os sinais subseqüentes do tipo especificado, ou SIG_DFL para restaurar a ação-padrão do sinal quando este ocorrer. A ação-padrão é eliminar qualquer processo ou ignorar o sinal, dependendo do sinal. Como um exemplo de como SIG_IGN é utilizado, considere o que acontece quando o *shell* cria um processo em segundo plano como resultado de

```
command &
```

seria indesejável que o sinal DEL do teclado afetasse o processo em segundo plano, então, depois de FORK mas antes de EXEC, o *shell* faz

```
sigaction(SIGINT, SIG_IGN, NULL);
```

e

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

para desabilitar os sinais DEL e QUIT. (O sinal de encerramento — *quit* — é gerado por CTRL-\; que é o mesmo

que DEL, exceto que se não é capturado nem ignorado, ele faz um *dump*[*] do processo eliminado.) Para processos em primeiro plano (sem &), esses sinais não são ignorados.

Pressionar a tecla DEL não é a única maneira de enviar um sinal. A chamada de sistema KILL permite que um processo sinalize outro processo (desde que eles tenham o mesmo *uid* — processos não-relacionados não podem sinalizar um para o outro). Voltando ao exemplo de processos em segundo plano utilizado acima, suponha que um processo em segundo plano seja iniciado, mas mais tarde decida-se que o processo deve ser terminado. SIGINT e SIGQUIT foram desativados, então algo mais é necessário. A solução é usar o programa *kill*, que usa a chamada de sistema KILL para enviar um sinal para qualquer processo. Enviando o sinal 9 (SIGKILL), para um processo em segundo plano, esse processo pode ser eliminado. SIGKILL não pode ser capturado nem ignorado.

Para muitos aplicativos de tempo real, um processo precisa ser interrompido após um intervalo específico de tempo para fazer algo, como retransmitir um pacote perdido em uma linha de comunicação pouco confiável. Para lidar com essa situação, a chamada de sistema ALARM foi fornecida. O parâmetro especifica um intervalo, em segundos, depois de que um sinal SIGALRM é enviado para o processo. Um processo só pode ter um alarme por vez. Se uma chamada ALARM é feita com um parâmetro de 10 segundos e, então, 3 segundos mais tarde outra chamada ALARM é feita com um parâmetro de 20 segundos, só um sinal será gerado 20 segundos depois da segunda chamada. O primeiro sinal é cancelado pela segunda chamada ALARM. Se o parâmetro ALARM for zero, qualquer sinal de alarme é cancelado. Se um sinal de alarme não for capturado a ação-padrão é tomada, e o processo sinalizado é eliminado.

Às vezes, ocorre que um processo não tem nada a fazer até que um sinal chegue. Por exemplo, considere um programa CAI (*Computer-Aided-Instruction* – ensino auxiliado por computador) que está testando velocidade de leitura e de entendimento. Ele exibe algum texto na tela e, então, chama ALARM para sinalizá-lo depois de 30 segundos. Enquanto o aluno estiver lendo o texto o programa não tem nada a fazer. Ele poderia entrar em um laço sem fazer nada, mas isso desperdiçaria tempo da CPU que outro processo ou usuário talvez precise. Uma idéia melhor é usar PAUSE, que instrui o MINIX a suspender o processo até o próximo sinal.

1.4.3 Chamadas de Sistema para Gerenciamento de Arquivos

Muitas chamadas de sistema estão relacionadas com o sistema de arquivos. Nesta seção examinaremos as chamadas que funcionam em arquivos individuais; na próxima, examinaremos as que envolvem diretórios ou o sistema de arquivos como um todo. Para criar um novo arquivo, a chamada CREAT é utilizada (o motivo pêlo qual a chamada é CREAT e não CREATE foi esquecido nas névoas do tempo). Seus parâmetros fornecem o nome do arquivo e o modo de proteção. Assim

```
fd = creat ("abc", 0751);
```

cria um arquivo chamado *abc* com modo 0751 octal (em C, um zero na frente significa que uma constante está em octal). Os 9 bits de ordem inferior de 0751 especificam os bits *rwx* para o proprietário (7 significa permissão para ler, gravar e para executar), seu grupo (5 significa permissão para ler e para executar) e outros (1 significa permissão só para executar).

CREAT não só cria um novo arquivo, mas, também, o abre para escrita, independente do modo do arquivo. O descritor de arquivo retornado, *fd*, pode ser utilizado para gravar o arquivo. Se um CREAT é executado em um arquivo existente, esse arquivo é truncado para comprimento 0, desde que, naturalmente, todas as permissões estejam corretas. A chamada CREAT é obsoleta, já que OPEN agora pode criar novos arquivos, mas ela foi incluída por razões de retrocompatibilidade.

Arquivos especiais são criados utilizando MKNOD em vez de CREAT. Uma típica chamada é

```
fd = mknod ("/dev/ttyc2", 020744, 0x0402);
```

que cria um arquivo chamado */dev/ttyc2* (o nome usual para o console 2) e lhe atribui o modo 020744 octal (um arquivo especial de caractere com bits de proteção *rwxr--r--*). O terceiro parâmetro contém o dispositivo principal (4) no byte de ordem superior e o dispositivo secundário (2) no byte de ordem inferior. O dispositivo principal poderia ser qualquer coisa, mas um arquivo chamado */dev/ttyc2* deve ser o dispositivo secundário 2. As chamadas para MKNOD falham a menos que a chamada tenha sido feita pelo superusuário.

Para ler ou para gravar um arquivo existente, o arquivo deve primeiro ser aberto usando OPEN. Essa chamada especifica o nome do arquivo a ser aberto, seja um nome de caminho absoluto ou relativo ao diretório de trabalho, e para um código de *O_RDONLY, O_WRONLY ou O_RDWR*, que significam aberto para leitura, para gravação ou para ambos. O descritor de arquivo retornado, então, pode ser utilizado para ler ou para gravar. Depois, o arquivo pode ser fechado por CLOSE, que torna o descritor de arquivo disponível para reutilização em um CREAT ou OPEN subseqüente.

As chamadas mais utilizadas são, sem dúvida, READ e WRITE. Vimos READ anteriormente. WRITE tem os mesmos parâmetros.

Embora a maioria dos programas leia e escreva arquivos seqüencialmente, alguns programas aplicativos precisam ser capazes de acessar qualquer parte de um arquivo aleatoriamente. Associado a cada arquivo, existe um ponteiro que indica a posição atual no arquivo. Quando lendo (gravando) seqüencialmente, normalmente ele aponta para o próximo byte a ser lido (gravado). A chamada LSEEK muda

[*] N. de R. *Dump*: Despejo binário. Parte da memória despejada em outro meio de armazenamento ou impressa em forma binária.

o valor da posição do ponteiro, de modo que chamadas subseqüentes READ ou WRITE podem começar em qualquer lugar no arquivo ou até além do fim.

LSEEK tem três parâmetros: o primeiro é o descritor de arquivo para o arquivo, o segundo é uma posição de arquivo e o terceiro informa se a posição do arquivo é relativa ao começo do arquivo, à posição atual ou ao fim do arquivo. O valor retornado por LSEEK é a posição absoluta no arquivo depois de mudar o ponteiro.

Para cada arquivo, o MINIX monitora o modo do arquivo (arquivo normal, arquivo especial, diretório e assim por diante), o tamanho, a data da última modificação e outras informações. Os programas podem pedir para ver essas informações via chamadas de sistema STAT e FSTAT. Essas diferem apenas no fato de que a primeira especifica o arquivo por nome, enquanto a última utiliza um descritor de arquivo, o que a torna útil para arquivos abertos, especialmente entrada-padrão e saída-padrão, cujos nomes podem não ser conhecidos. Ambas as chamadas fornecem como segundo parâmetro um ponteiro para uma estrutura onde as informações estão definidas. A estrutura é mostrada na Figura 1-12.

Ao manipular descritores de arquivo, a chamada DUP é ocasionalmente útil. Considere, por exemplo, um programa que precise fechar a saída-padrão (descritor de arquivo 1), substituir outro arquivo como saída-padrão, chamar uma função que grava uma saída qualquer na saída-padrão e, então, restaurar a situação original. Basta fechar o descritor de arquivo 1 e, então, abrir um novo arquivo para que o novo arquivo torne-se a saída-padrão (supondo que a entrada-padrão, o descritor de arquivo 0, está em uso), mas será impossível restaurar a situação original mais tarde.

A solução é primeiro executar a declaração

```
fd = dup(1);
```

que utiliza a chamada de sistema DUP para alocar um novo descritor de arquivo, *fd*, e arranjar para que ele corresponda ao mesmo arquivo que a saída-padrão. A saída-padrão, então, pode ser fechada e um novo arquivo ser aberto e utilizado. Quando chegar o momento de restaurar a situação original, o descritor de arquivo 1 pode ser fechado e, então,

```
n = dup (fd);
```

executado para atribuir o descritor de arquivo mais baixo, a saber, 1, para o mesmo arquivo que *fd*. Por fim, *fd* pode ser fechado e voltamos ao ponto aonde começamos.

A chamada DUP tem uma variante que permite a um descritor de arquivo arbitrário não-atribuído ser criado para referir-se a um dado arquivo aberto. Ele é chamado por

```
dup2(fd, fd2);
```

onde *fd* referencia um arquivo aberto e *fd2* é o descritor de arquivo não-atribuído que é criado para referenciar o mesmo arquivo que *fd*. Assim, se *fd* referenciar a entrada-padrão (o descritor de arquivo 0) e *fd2* for 4, após a chamada, os descritores de arquivo 0 e 4 irão referenciar a entrada-padrão.

A comunicação interprocesso no MINIX usa *pipes*, como descrito anteriormente. Quando um usuário digita

```
cat file1 file2 | sort
```

o *shell* cria um *pipe* e prepara a saída-padrão do primeiro processo gravando no *pipe* e, assim, a entrada-padrão do segundo processo pode ler a partir dele. A chamada de sistema PIPE cria um *pipe* e retorna dois descritores de arquivo, um para gravar e outro para ler. A chamada é

```
pipe (&fd[0]);
```

onde *fd* é uma matriz de dois números inteiros e *fd*[0] é o descritor de arquivo para ler e *fd*[1] para gravar. Geralmente, um FORK vem em seguida e o pai fecha o descritor de arquivo para ler, e o filho fecha o descritor de arquivo para gravar (ou vice-versa), assim um processo pode ler o *pipe* e o outro pode gravar nele.

A Figura 1-13 representa um esqueleto do procedimento que cria dois processos, com a saída do primeiro canalizado para o segundo. (Um exemplo mais realista faria verificação de erro e manipularia argumentos.) Primeiro um *pipe* é criado e, então, o procedimento ramifica-se, eventualmente com o pai tornando-se o primeiro processo no *pi-*

```
struct stat {
    short st_dev;                  /* dispositivo a quem o nó-i pertence */
    unsigned short st_ino;         /* número de nó-i */
    unsigned short st_mode;        /* modo palavra */
    short st_mink;                 /* número de links */
    short st_uid;                  /* id do usuário */
    short st_gid;                  /* id do grupo */
    short st_rdev;                 /* dispositivo principal/secundário para arquivos especiais
    long st_size;                  /* tamanho do arquivo */
    long st_atime;                 /* data do último acesso */
    long st_mtime;                 /* data da última modificação */
    long st_ctime;                 /* data da última alteração no nó-i */
};
```

Figura 1-12 A estrutura utilizada para retornar as informações para chamadas de sistema STAT e FSTAT. No código real, nomes simbólicos são utilizados para alguns tipos.

```
#define STD_INPUT 0                      /* descritor de arquivo para a entrada-padrão */
#define STD_OUTPUT 1                     /* descritor de arquivo para a saída-padrão */

pipeline(process1, process2)
char *process1, *process2;               /* ponteiros para nomes de programa */
{
 int fd[2];

 pipe(&fd[0]);                           /* cria um pipe */
 if (fork() != 0) {
     /* O processo pai executa estas declarações. */
     close(fd[0]);                       /* o processo 1 não precisa ler do pipe */
     close(STD_OUTPUT);                  /* prepara para nova saída-padrão */
     dup(fd[1]);                         /* saída-padrão definida como fd[1] */
     close(fd[1]);                       /* este descritor de arquivo não é mais necessário */
     execl(process1, process1, 0);
 } else {
     /* O processo-filho executa estas declarações. */
     close(fd[1]);                       /* o processo 2 não necessita gravar no pipe */
     close(STD_INPUT);                   /* prepara para nova entrada-padrão */
     dup(fd[0]);                         /* entrada-padrão definida como fd [0] */
     close(fd[0]);                       /* este descritor de arquivo não é mais necessário */
     execl(process2, process2, 0);
 }
}
```

Figura 1-13 Um esqueleto para configurar um *pipeline* de dois processos.

peline e o processo-filho tornando-se o segundo. Como os arquivos a serem executados, *process1* e *process2*, não sabem que são partes de um *pipeline**, é essencial que o descritor de arquivo seja tratado de modo que a saída-padrão do primeiro processo seja o *pipe* e a entrada-padrão do segundo seja o *pipe*. O pai primeiro fecha o descritor de arquivo para leitura do *pipe*. Então, ele fecha a saída-padrão e faz uma chamada DUP que permite que o descritor de arquivo 1 grave no *pipe*. É importante saber que DUP sempre retorna o descritor de arquivo mais baixo disponível, neste caso, 1. Então, o programa fecha o outro descritor de arquivo do *pipe*.

Depois da chamada EXEC, o processo iniciado terá descritores de arquivo 0 e 2 inalterados, e o descritor de arquivo 1 para gravar no *pipe*. O código-filho é análogo. O parâmetro para *execl* é repetido porque o primeiro é o arquivo a ser executado e o segundo é o primeiro parâmetro, que a maioria dos programas espera que seja o nome de arquivo.

A próxima chamada de sistema, IOCTL, é potencialmente aplicável a todos os arquivos especiais. Ela é utilizada, por exemplo, por *drivers* de dispositivo de bloco como o *driver* SCSI para controlar dispositivos de fita e de CD-ROM. Entretanto, seu uso principal é com arquivos de caractere especiais, principalmente terminais. O POSIX define diversas funções que a biblioteca traduz em chamadas IOCTL. As funções de biblioteca *tcgetattr* e *tcsetattr* utilizam IOCTL para mudar os caracteres utilizados para corrigir erros de digitação no terminal, mudar o modo do terminal, etc.

*N. de R. *Pipeline*: canalização. É o canal de comunicação criado por um *pipe* ligando dois processos.

Modo processado é o modo terminal normal, no qual os caracteres de apagamento e de eliminação trabalham normalmente, CTRL-S e CTRL-Q podem ser utilizados para parar e para iniciar a saída de terminal, CTRL-D significa fim de arquivo, DEL gera um sinal de interrupção e CTRL-\ gera um sinal de encerramento para forçar um *dump* de núcleo.

No **modo bruto**, todas essas funções são desativadas; cada caractere é passado diretamente para o programa sem nenhum processamento especial. Além disso, no modo bruto, uma leitura a partir do terminal dará para o programa quaisquer caracteres que foram digitados, até mesmo uma linha parcial, em vez de esperar uma linha completa ser digitada, como no modo processado.

O **modo Cbreak** é o meio-termo. Os caracteres de apagamento e de eliminação como CTRL-D são desativados para edição, mas CTRL-S, CTRL-Q, DEL e CTRL-\ são ativados. Como no modo bruto, linhas parciais podem ser retornadas ao programa (se a edição entre linhas estiver desativada, não há necessidade de esperar até que uma linha inteira seja recebida — o usuário não pode mudar de idéia e excluí-la, como pode no modo processado).

O POSIX não usa os termos processado, bruto e *cbreak*. Na terminologia do POSIX, **modo canônico** corresponde ao modo processado. Nesse modo, há 11 caracteres especiais, e a entrada é por linhas. No **modo não-canônico**, um número mínimo de caracteres a receber e uma medida de tempo, especificada em unidades de 1/10 segundo, determina como uma leitura será satisfeita. Sob o POSIX, há muita flexibilidade e vários sinalizadores podem ser definidos para fazer o modo não-canônico comportar-se como

modo *cbreak* ou modo bruto. Os termos antigos são mais descritivos e continuaremos a usá-los informalmente.

IOCTL tem três parâmetros, por exemplo, uma chamada para *tcsetattr* para definir parâmetros terminais resultará em

```
ioctl (fd, TCSETS, &termios);
```

o primeiro parâmetro especifica um arquivo, o segundo, uma operação e o terceiro é o endereço da estrutura do POSIX que contém sinalizadores e a matriz de caracteres de controle. Outros códigos de operação podem adiar as mudanças até que toda saída tenha sido enviada, fazer com que uma entrada não lida seja descartada e retornar os valores atuais.

A chamada de sistema ACESS é utilizada para determinar se um certo acesso a arquivo é permitido pelo sistema de proteção. Ela é necessária, porque alguns programas podem executar usando um *uid* de um usuário diferente. Esse mecanismo de SETUID será descrito mais tarde.

A chamada de sistema RENAME é utilizada para dar um novo nome a um arquivo. Os parâmetros especificam o nome antigo e o novo.

Por fim, a chamada FCNTL é utilizada para controlar arquivos, mais ou menos análoga à IOCTL (i. e., ambas são *hacks** horríveis). Ela tem várias opções, a mais importante das quais é para bloqueio de arquivo para consulta. Utilizando FCNTL, é possível para um processo bloquear e desbloquear partes de arquivos e testar parte de um arquivo para ver se ele está bloqueado. A chamada não impõe qualquer semântica de bloqueio. Os programas devem fazer isso por si mesmos.

1.4.4 Chamadas de Sistema para Gerenciamento de Diretórios

Nesta seção examinaremos algumas chamadas de sistema que se relacionam mais com diretórios ou com o sistema de arquivos como um todo, em vez de somente com um arquivo específico como na seção anterior. As primeiras duas chamadas, MKDIR e RMDIR, criam e removem diretórios vazios, respectivamente. A próxima chamada é LINK. Seu propósito é permitir que o mesmo arquivo apareça sob dois ou mais nomes, freqüentemente em diretórios diferentes. Um uso típico é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, com cada um deles tendo o arquivo aparecendo no seu próprio diretório, possivelmente sob nomes diferentes. Compartilhar um arquivo não é o mesmo que dar a cada membro de equipe uma cópia privada, porque ter um arquivo compartilhado significa que alterações feitas por qualquer membro da equipe são instantaneamente visíveis para os outros membros — há só um arquivo. Quando são feitas cópias de um arquivo, as mudanças subseqüentes feitas em uma cópia não têm efeito nas outras.

Para ver como LINK funciona, considere a situação da Figura 1-14(a). Aqui estão dois usuários, *ast* e *jim*, cada um tendo seus próprios diretórios com alguns arquivos. Se *ast* agora executa um programa que contém a chamada de sistema

```
link("/usr/jim/memo", "/usr/ast/note");
```

o arquivo *memo* no diretório de *jim* agora está inserido no diretório de *ast* sob o nome *note*. Portanto, */usr/jim/memo* e */usr/ast/note* referem-se ao mesmo arquivo.

O entendimento de como LINK funciona provavelmente tornará mais claro o que ele faz. Cada arquivo no MINIX tem um número único, seu número i, que o identifica. Tal número-i é um índice em uma tabela de **nós-i**, um por arquivo, informando quem é o proprietário do arquivo, onde seus blocos de disco estão e assim por diante. Um diretório é simplesmente um arquivo que contém um conjunto de pares (número-i, nome em ASCII). Na Figura 1-14, *mail* tem 16 como número-i e assim por diante. O que LINK faz é simplesmente criar uma nova entrada de diretório com um (possivelmente novo) nome, usando o número-i de um arquivo existente. Na Figura 1-14(b), duas entradas têm o mesmo número-i (70) e assim referenciam o mesmo arquivo. Se qualquer uma mais tarde for removida, usando a chamada de sistema UNLINK, o outro permanece. Se ambas forem removidas, o MINIX vê que nenhuma entrada para o arquivo existe (um campo no nó-i monitora o número de entradas de diretório que apontam para o arquivo), então o arquivo é removido do disco.

Como mencionamos anteriormente, a chamada de sistema MOUNT permite que dois sistemas de arquivos fundam-se em um. Uma situação comum é ter o **sistema de arquivos-raiz,** contendo as versões binárias (executáveis) dos comandos comuns e outros arquivos intensamente utilizados, no disco de RAM. O usuário, então, pode inserir um disquete, por exemplo, que contém programas de usuário, na unidade 0.

Executando a chamada de sistema MOUNT, o sistema de arquivos da unidade 0 pode ser anexado ao sistema de arquivos-raiz, como mostrado na Figura 1-15. Uma típica declaração em C para executar a montagem é

```
mount("/dev/fd0", "/mnt", 0);
```

onde o primeiro parâmetro é o nome de um arquivo de bloco especial para a unidade 0 e o segundo parâmetro é o lugar na árvore onde ele será montado.

Depois da chamada MOUNT, um arquivo na unidade 0 pode ser acessado usando apenas seu caminho a partir do diretório-raiz ou do diretório de trabalho, sem considerar em qual unidade ele está. Aliás, a segunda, a terceira e a quarta unidades também podem ser montadas em qualquer lugar na árvore. O comando MOUNT torna possível integrar mídia removível em uma única hierarquia integrada de arquivos, sem precisar preocupar-se com o dispositivo em que um arquivo está. Embora esse exemplo en-

*N. de T. *Hack* – **1.** Modificação feita no código de um programa, em geral, sem que se dedique o tempo necessário para encontrar uma solução elegante. **2.** trabalho malfeito. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

Figura 1-14 (a) Dois diretórios antes de vincular */usr/jim/memo* com o diretório ast. (b) Os mesmos diretórios depois da vinculação.

volva disquetes, também podem ser montados desta maneira discos rígidos ou partes de discos rígidos (freqüentemente chamadas **partições** ou **dispositivos secundários**). Quando um sistema de arquivos não é mais necessário, ele pode ser desmontado com a chamada de sistema UMOUNT.

O MINIX mantém um *cache* de blocos recentemente utilizados na memória principal para evitar precisar lê-los do disco se eles forem utilizados de novo rapidamente. Se um bloco no *cache* é modificado (por um WRITE em um arquivo) e o sistema falhar antes do bloco modificado ser gravado no disco, o sistema de arquivos será danificado. Para limitar o dano potencial, é importantes descarregar o *cache* periodicamente, de modo que a quantidade de dados perdidos por causa de uma falha será pequena. A chamada de sistema SYNC informa o MINIX para gravar todos os blocos de *cache* que foram modificados desde que eles foram lidos. Quando o MINIX é iniciado, um programa chamado *update* é iniciado como um processo de segundo plano para fazer um SYNC a cada 30 segundos, mantendo o *cache* limpo.

Duas outras chamadas relacionadas com diretórios são CHDIR e CHROOT. A primeira muda o diretório de trabalho e a última muda o diretório raiz. Depois da chamada

```
chdir ("/usr/ast/test");
```

uma chamada open para o arquivo *xyz* abrirá */usr/ast/test/xyz*. CHROOT funciona de maneira análoga. Uma vez que um processo informou o sistema para mudar seu diretório raiz, todo nome de caminho absoluto (nomes de caminho começam com um "/") iniciará na nova raiz. Só superusuários podem executar CHROOT e superusuários regulares não o fazem com muita freqüência.

1.4.5 Chamadas de Sistema para Proteção

No MINIX cada arquivo tem um modo de 11 bits utilizado para proteção, nove dos quais são os bits de leitura-gravação-execução para o proprietário, para o grupo e para outros. A chamada de sistema CHMOD torna possível mudar o modo de um arquivo. Por exemplo, para tornar somente de leitura um arquivo para todos, exceto o proprietário, pode-se executar

```
chmod ("file", 0644);
```

Os outros dois bits de proteção, 02000 e 04000, são os bits de SETGID (*set-group-id*) e SETUID (*set-user-id*), respectivamente. Quando qualquer usuário executa um programa com o bit de SETUID ativado, até o fim desse processo o *uid* efetivo do usuário é mudado para o do proprietário do arquivo. Esse recurso é intensamente utilizado para permitir que os usuários executem programas que realizam funções exclusivas do superusuário, como criar diretórios. A criação de um diretório utiliza MKNOD, que é exclusiva do superusuário. Arranjando para o programa *mkdir* ser possuído pelo superusuário e ter o modo 04755, os usuários normais podem ter o poder de executar MKNOD, mas de um modo bastante restrito.

Quando um processo executa um arquivo que tem o bit de SETUID ou SETGID em seu modo, ele adquire um *uid* ou *gid* efetivo diferente de seu *uid* ou *gid* real. Às vezes, ele é importante para um processo saber qual é seu *uid* ou *gid* efetivo e real. As chamadas de sistema GETUID e GETGID foram fornecidas para proporcionar essas informações. Cada chamada retorna o *uid* ou *o gid* efetivo e real, então quatro rotinas de biblioteca são necessárias para extrair as in-

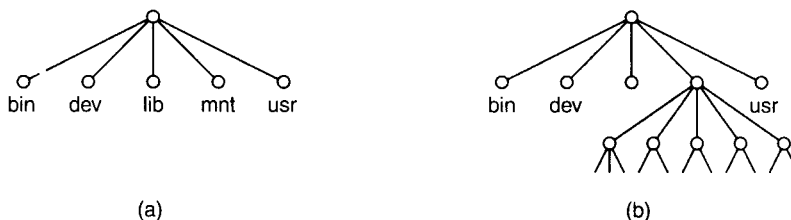



Figura 1-15 (a) Sistema de arquivos antes da montagem. (b) Sistema de arquivos depois da montagem.

formações adequadas: *getuid*, *getgid*, *geteuid* e *getegid*. As duas primeiras recebem o *uid/gid* real e as últimas duas os efetivos.

Usuários normais não podem mudar seu *uid*, exceto executando programas com o bit de SETUID ativado, mas o superusuário tem outra possibilidade: a chamada de sistema SETUID, que define os *uids* real e efetivo. SETGID define os dois *gids*. O superusuário também pode mudar o proprietário de um arquivo com a chamada de sistema CHOWN. Em resumo, o superusuário tem várias oportunidades para transgredir todas as regras de proteção, o que explica por que tantos estudantes dedicam tanto de seu tempo a tentar tornar-se superusuário.

As últimas duas chamadas de sistema nesta categoria podem ser executadas por processos de usuários normais. A primeira, UMASK, define uma máscara interna de bits para o sistema, que é utilizada para mascarar bits de modo quando um arquivo é criado. Após a chamada

```
umask(022);
```

o modo fornecido por CREAT e MKNOD terá os bits 022 mascarados antes de serem utilizados. Assim a chamada

```
creat ("file", 0777);
```

definirá o modo para 0755 em vez de 0777. Como a máscara de bit é herdada por processos-filho, se o *shell* fizer um UMASK imediatamente após o *login*, nenhum dos processos de usuário nessa sessão criarão acidentalmente arquivos em que outras pessoas podem gravar.

Quando um programa possuído pela raiz tem o bit de SETUID ativado, ele pode acessar qualquer arquivo, porque seu *uid* efetivo é o de superusuário. Freqüentemente é útil para o programa saber se a pessoa que chamou o programa tem permissão para acessar um determinado arquivo. Se o programa tentar somente o acesso, ele sempre será bem-sucedido e, portanto, não perceberá nada.

O que é necessário é uma maneira de ver se o acesso é permitido para o *uid* real. A chamada de sistema ACCESS fornece uma maneira de sabê-lo. O parâmetro *mode* é 4 para verificar acesso de leitura, 2 para acesso de gravação e 1 para acesso de execução. As combinações também são permitidas, por exemplo, com *mode* igual a 6, a chamada retorna 0 se são permitidos acesso de leitura e gravação para o *uid* real; caso contrário, -1 é retornado. Com *mode* igual a 0, uma verificação é feita para ver se o arquivo existe e se os diretórios que levam até ele podem ser pesquisados.

1.4.6 Chamadas de Sistema para Gerenciamento de Tempo

O MINIX tem quatro chamadas de sistema que envolvem o tempo de relógio convencional. TIME retorna a hora atual em segundos, com 0 correspondendo a 1° de Jan. de 1970 à meia-noite (exatamente quando o dia está iniciando, não quando está acabando). Naturalmente, o relógio do sistema deve ser configurado em algum ponto para permitir ser lido mais tarde, então STIME foi fornecida para permitir que o relógio seja configurado (pelo superusuário). A terceira chamada de tempo é UTIME, que permite que o proprietário de um arquivo (ou o superusuário) mude a data/hora armazenada no nó-i do arquivo. A aplicação desta chamada de sistema é relativamente limitada, mas alguns programas precisam dela como, por exemplo, *touch*, que configura a data/hora do arquivo como a data/hora atual.

Por fim, temos TIMES, que retorna as informações de contagem de um processo, de tal modo que se possa ver quanto tempo de CPU ele tem utilizado diretamente e quanto tempo de CPU o sistema em si gastou em seu favor (manipulando suas chamadas de sistema). Os tempos de usuário e de sistema totais utilizados por todos os seus filhos combinados também são retornados.

1.5 A ESTRUTURA DO SISTEMA OPERACIONAL

Agora que vimos como os sistemas operacionais se parecem externamente (i. e., a interface do programador), é hora de vê-los por dentro. Nas seções a seguir, examinaremos quatro estruturas diferentes que foram experimentadas, a fim de obter uma idéia do espectro de possibilidades. Essas não são de modo algum exaustivas, mas dão uma idéia de alguns modelos que foram experimentados na prática. Os quatro modelos são sistemas monolíticos, sistemas em camadas, máquinas virtuais e sistemas cliente-servidor.

1.5.1 Sistemas Monolíticos

Esta é, de longe, a organização mais comum. Esta abordagem poderia muito bem ser subintitulada "A Grande Bagunça". A estrutura é tal que não há nenhuma estrutura. O sistema operacional é escrito como uma coleção de procedimentos, cada um dos quais pode chamar qualquer um dos outros sempre que precisar. Quando essa técnica é utilizada, cada procedimento no sistema tem uma interface bem-definida em termos de parâmetros e de resultados e cada um é livre para chamar qualquer um dos outros, se o último fornecer alguma computação útil de que o primeiro precisa.

Para construir o programa-objeto do sistema operacional, quando essa aproximação é utilizada, primeiro deve-se compilar todos os procedimentos ou os arquivos individuais que contêm os procedimentos e, então, agrupá-los todos juntos em um único arquivo-objeto usando o *linkeditor** do sistema. Em termos de proteção de informações, não há essencialmente nenhuma — cada procedimento é visível para todos os demais (em oposição a uma estrutura contendo módulos ou pacotes, em que muitas das informa-

*N. de R. *Linkeditor*: programa que reúne arquivos-objeto compilados independentemente, de modo a gerar um arquivo executável. Este processo também é referido como vinculação ou montagem ao longo deste livro.

ções são ocultas dentro de módulos e só os pontos de entrada oficialmente designados podem ser chamados de fora do módulo).

Mesmo em sistemas monolíticos, entretanto, é possível ter pelo menos um pouco de estrutura. Os serviços (chamadas de sistema) fornecidos pelo sistema operacional são requisitados — colocando-se os parâmetros em lugares bem definidos, como em registradores ou na pilha e, então, executando uma instrução especial de interrupção conhecida como **chamada de *kernel*** ou **chamada de supervisor.**

Essa instrução comuta a máquina do modo usuário para modo *kernel* e transfere o controle para o sistema operacional, mostrado como o evento (1) na Figura 1-16. (A maioria das CPUs tem dois modos: modo *kernel*, para o sistema operacional, em que todas as instruções são permitidas, e modo usuário, para programas de usuário, nos quais a E/S e outras instruções não são permitidas.)

O sistema operacional, então, examina os parâmetros da chamada para determinar qual chamada de sistema deve ser executada, mostrado como (2) na Figura 1-16. Em seguida, o sistema operacional pesquisa em uma tabela que contém em uma entrada *k* um apontador para o procedimento que executa a chamada de sistema *k*. Essa operação, mostrada como (3) na Figura 1-16, identifica o procedimento do serviço, que, então, é chamado. Quando o trabalho é completado e a chamada de sistema acaba, o controle é devolvido para o programa de usuário (passo 4), de tal modo que ele pode continuar a execução com a declaração que se segue à chamada de sistema. Essa organização sugere uma estrutura básica para o sistema operacional:

1. Um programa principal que invoca o procedimento de serviço requisitado.
2. Um conjunto de procedimentos de serviços que executa as chamadas de sistema.
3. Um conjunto de procedimentos utilitários que ajuda os procedimentos de serviços.

Nesse modelo, para cada chamada de sistema há um procedimento de serviço que cuida dela. Os procedimentos utilitários fazem coisas que são necessárias para vários procedimentos de serviço, como buscar dados de programas de usuário. Essa divisão dos procedimentos em três camadas é mostrada na Figura 1-17.

1.5.2 Sistemas em Camadas

Uma generalização da abordagem da Figura 1-17 é organizar o sistema operacional como uma hierarquia de camadas, construídas uma sobre a outra. O primeiro sistema construído dessa maneira foi o sistema criado no Technische Hogeschool Eindhoven, na Holanda, por E. W. Dijkstra (1968) e seus alunos. O sistema THE era um sistema de lote simples para um computador holandês, o Electrologica X8 , que tinha 32K de palavras de 27 bits (bits eram caros naquela época).

O sistema tinha seis camadas, como mostrado na Figura 1-18. A camada 0 lidava com a alocação do processador, alternando entre processos quando ocorriam interrupções ou quando temporizadores expiravam. Acima da camada 0, o sistema consistia em processos seqüenciais, cada um dos quais podia ser programado sem ser necessário preocupar-se com o fato de que múltiplos processos estavam executando num único processador. Em outras palavras, a camada 0 proporcionava a multiprogramação básica da CPU.

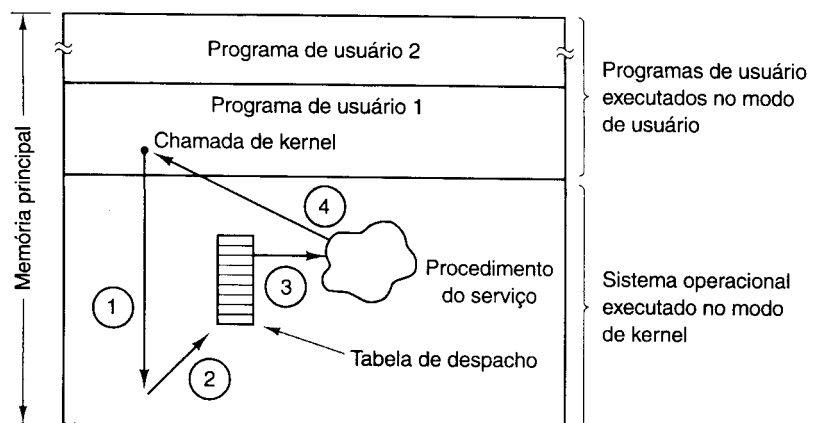



Figura 1-16 Como uma chamada de sistema pode ser feita: (1) O programa de usuário gera uma interrupção para o *kernel*. (2) O sistema operacional determina o número do serviço necessário. (3) O sistema operacional chama o procedimento de serviço. (4) O controle é retornado para o programa de usuário.

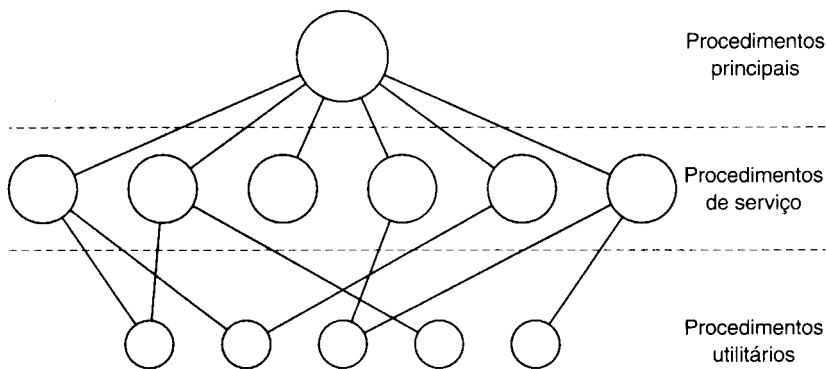



Figura 1-17 Um modelo simples de estruturação para um sistema monolítico.

A camada 1 fazia o gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor* com 512K de palavras utilizado para armazenar partes dos processos (páginas) para os quais não havia lugar na memória principal. Acima da camada 1, processos não tinham que se preocupar com o fato de eles estarem em memória ou no tambor; o software da camada 1 cuidava de assegurar que as páginas fossem levadas para a memória sempre que fossem necessárias.

A camada 2 manipulava a comunicação entre cada processo e o console do operador. Acima dessa camada, cada processo efetivamente tinha seu próprio console de operador. A camada 3 cuidava de gerenciar os dispositivos de E/S e de armazenar os fluxos de informação para eles e a partir deles. Acima da camada 3, cada processo podia lidar com dispositivos abstratos de E/S com propriedades amigáveis, em vez de dispositivos reais com muitas peculiaridades. A camada 4 era onde os programas de usuário localizavam-se. Eles não tinham de preocupar-se com gerenciamento de processos, de memória, de console ou de E/S. O processo de operador do sistema localizava-se na camada 5.

Uma posterior generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, o MULTICS foi organizado como uma série de anéis concêntricos, com os internos sendo mais privilegiados do que os externos. Quando um procedimento em um anel externo queria chamar um procedimento em um anel interno, ele tinha de fazer o equivalente de uma chamada de sistema, isto é, uma instrução TRAP cujos parâmetros eram cuidadosamente verificados quanto à validade antes de permitir que a chamada prosseguisse. Embora o sistema operacional inteiro fosse parte do espaço de endereçamento de cada processo de usuário no MULTICS, o hardware tornava possível designar procedimentos individuais (segmentos de me-

Camada	Função
5	Operador
4	Programa usuário
3	Gerenciamento de entrada/saída
2	Comunicação operador-processo
1	Gerenciamento de memória e tambor
0	Alocação do processador e multiprogramação

Figura 1-18 A estrutura do sistema operacional THE.

*N. de R. Antigo meio magnético de armazenamento de dados.

mória, na realidade) como protegidos contra leitura, gravação ou execução.

Enquanto o esquema em camadas do THE era realmente só um auxílio para modelagem, porque todas as partes do sistema estavam em última instância vinculadas juntas em um único programa objeto, no MULTICS o mecanismo de anel era muito presente em tempo de execução e reforçado pelo hardware. A vantagem do mecanismo de anel é que ele podia facilmente ser estendido para estruturar subsistemas de usuário. Por exemplo, um professor podia escrever um programa para testar e para avaliar programas de aluno e executar esse programa no anel n, com o programa do aluno sendo executado no anel $n + 1$ de tal modo que eles não podiam mudar seus níveis.

1.5.3 Máquinas Virtuais

As versões iniciais do OS/360 eram estritamente sistemas de lote. Não obstante, muitos usuários do 360 queriam ter tempo compartilhado, assim vários grupos, tanto de dentro como de fora da IBM decidiram escrever sistemas de tempo compartilhado para ele. O sistema oficial de tempo compartilhado da IBM, o TSS/360, foi lançado tardiamente; quando finalmente chegou, era tão grande e lento que poucos ambientes foram convertidos para ele. Por fim, ele acabou sendo abandonado depois que seu desenvolvimento tinha consumido algo em torno de US$ 50 milhões (Graham, 1970). Mas um grupo no Centro Científico da IBM em Cambridge, Massachusetts, produziu um sistema radicalmente diferente que a IBM acabou aceitando como um produto e que agora é amplamente utilizado em seus *mainframes* remanescentes.

Esse sistema, originalmente chamado CP/CMS e mais tarde rebatizado como VM/370 (Seawright e MacKinnon, 1979), foi baseado em uma observação astuta: um sistema de tempo compartilhado fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente que o hardware básico. A essência do VM/370 foi separar completamente essas duas funções.

O coração do sistema, conhecido como **monitor de máquina virtual,** rodava no hardware básico e fazia a multiprogramação, oferecendo não uma, mas várias máquinas virtuais à camada superior seguinte, como mostrado na Figura 1-19. Entretanto, ao contrário de todos os outros sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e com outros recursos amigáveis. Em vez disso, elas são cópias *exatas* do hardware básico, incluindo o modo *kernel*/usuário, E/S, interrupções e tudo mais que uma máquina real tem.

Como cada máquina virtual é idêntica ao hardware verdadeiro, cada uma pode executar qualquer sistema operacional que executará diretamente sobre o hardware básico. Máquinas virtuais diferentes podem e freqüentemente executam sistemas operacionais diferentes. Algumas executam um dos descendentes do OS/360 para processamento de transações ou de lotes, enquanto outras executam um sistema interativo monousuário chamado CMS (*Conversational Monitor System*) para usuários de tempo compartilhado.

Quando um programa CMS executa uma chamada de sistema, a chamada é interceptada pelo sistema operacional da sua própria máquina virtual, não pelo VM/370, exatamente como faria se estivesse executando em uma máquina real em vez de em uma virtual. O CMS, então, emite as instruções normais de E/S de hardware para ler seu disco virtual, ou o que é necessário para executar a chamada. Essas instruções de E/S são interceptadas pelo VM/370, que então as executa como parte de sua simulação do hardware real. Fazendo uma separação completa das funções de multiprogramação e oferecendo uma máquina estendida, cada um dos pedaços pode ser muito mais simples, mais flexível e mais fácil de manter.

A idéia de uma máquina virtual é intensamente utilizada hoje em dia em um contexto diferente: executando programas antigos de MS-DOS em um Pentium (ou outra CPU de 32 bits da Intel). Ao projetar o Pentium e seu software, tanto a Intel como a Microsoft reconheceram que haveria uma grande demanda para executar software antigo no novo hardware. Por essa razão, a Intel ofereceu um modo virtual 8086 no Pentium. Assim, a máquina age como um 8086 (que é idêntico a um 8088 do ponto de vista do software), incluindo o endereçamento de 16 bits com um limite de 1MB.

Este modo é utilizado pelo WINDOWS, o OS/2 e outros sistemas operacionais para executar programas do MS-DOS. Esses programas são iniciados no modo 8086 virtual. Contanto que executem instruções normais, eles rodam sobre o hardware básico. Entretanto, quando um programa ten-

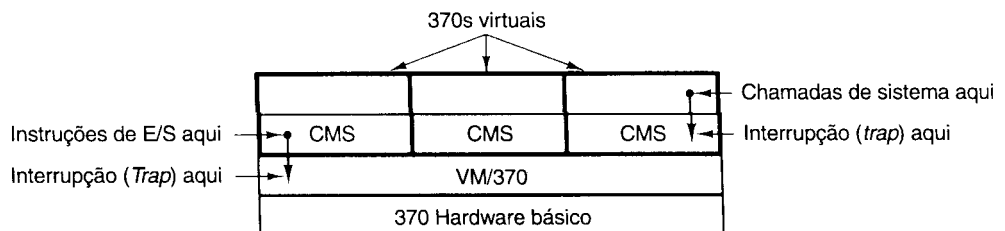



Figura 1-19 A estrutura do VM/370 com CMS.

ta interceptar o sistema operacional para fazer uma chamada de sistema ou tenta fazer E/S protegida diretamente, ocorre uma interrupção para o monitor da máquina virtual.

Duas variantes nesse modelo são possíveis. Na primeira, o próprio MS-DOS é carregado no espaço de endereço virtual do 8086, de tal modo que o monitor da máquina virtual somente reflete de volta a interrupção para o MS-DOS, assim como aconteceria em um 8086 real. Quando o MS-DOS mais tarde tenta fazer a E/S sozinho, essa operação é capturada e executada pelo monitor da máquina virtual.

Na outra variante, o monitor da máquina virtual captura a primeira interrupção e faz a E/S sozinho, desde que saiba quais são todas as chamadas de sistema do MS-DOS e, portanto, saiba o que cada interrupção deve fazer. Essa variante é menos pura do que a primeira, já que emula apenas o MS-DOS corretamente e não outros sistemas operacionais, como faz a primeira. Por outro lado, ela é muito mais rápida, uma vez que elimina o problema de iniciar o MS-DOS para fazer a E/S. Uma desvantagem de realmente executar o MS-DOS no modo 8086 virtual é que o MS-DOS desperdiça muito tempo às voltas com a habilitação de interrupções, processo que deve ser emulado a um custo significativo.

Vale notar que nenhuma dessas abordagens é realmente a mesma do VM/370, uma vez que a máquina sendo emulada não é um Pentium completo, mas apenas um 8086. Com o sistema VM/370, é possível executar o VM/370 em si na máquina virtual. Com o Pentium, não é possível executar, digamos, WINDOWS no 8086 virtual porque nenhuma versão do WINDOWS roda em um 8086; um 286 é o mínimo mesmo para a versão mais antiga, e a emulação de 286 não é fornecida (fica só a emulação do Pentium).

Com o VM/370, cada processo de usuário recebe uma cópia exata do computador real. Com o modo 8086 virtual no Pentium, cada processo de usuário recebe uma cópia exata de um computador diferente. Dando um passo à frente, pesquisadores do M.I.T. construíram um sistema que dá um clone do computador real para cada usuário, mas com um subconjunto dos recursos (Engler *et al.*, 1995). Assim, uma máquina virtual poderia receber os blocos de disco de 0 a 1023, a seguinte poderia receber os blocos de 1024 a 2047 e assim por diante.

Na camada mais inferior, executando em modo de *kernel*, está um programa chamado ***exokernel***. Seu trabalho é atribuir recursos a máquinas virtuais e, então, verificar tentativas de utilizá-los para assegurar-se de que nenhuma máquina está tentando usar recursos de outra. Cada máquina virtual no nível do usuário pode executar o próprio sistema operacional, como no VM/370 e nos 8086s virtuais do Pentium, exceto que cada uma é limitada a usar só os recursos que ela solicitou e que lhe foram alocados.

A vantagem do esquema de *exokernel* é que ele economiza uma camada de mapeamento. Em outros projetos, cada máquina virtual pensa que tem seu próprio disco, com blocos que executam de 0 até algum máximo, assim o monitor de máquina virtual deve manter tabelas para remapear endereços de disco (e todos os outros recursos). Com o *exokernel*, esse remapeamento não é necessário. O *exokernel* precisa apenas monitorar qual recurso foi designado a qual máquina virtual. Esse método mantém a vantagem de separar a multiprogramação (no *exokernel*) do código do sistema operacional de usuário (no espaço do usuário), mas com menor sobrecarga, uma vez que tudo o que o *exokernel* tem de fazer é manter as máquinas virtuais separadas.

1.5.4 Modelo Cliente-Servidor

O VM/370 ganha muito em simplicidade, movendo uma parte grande do código tradicional do sistema operacional (implementando a máquina estendida) para uma camada mais alta, CMS. Entretanto, o VM/370 em si continua sendo um programa complexo porque simular diversos 370s virtuais não é *tão* simples (especialmente se você quiser fazê-lo de maneira razoavelmente eficiente).

Uma tendência nos sistemas operacionais modernos é levar mais adiante ainda essa idéia de mover código para camadas mais altas e remover tanto quanto possível do sistema operacional, deixando um mínimo de *kernel*. A abordagem normal é implementar a maior parte das funções do sistema operacional em processos de usuário. Para requisitar um serviço, como ler um bloco de um arquivo, um processo de usuário (agora conhecido como **processo cliente**) envia a requisição para um **processo servidor**, que, então, faz o trabalho e remete de volta a resposta.

Nesse modelo, mostrado na Figura 1-20, tudo que o *kernel* faz é gerenciar a comunicação entre clientes e servidores. Dividir o sistema operacional em partes, cada uma gerenciando apenas uma faceta do sistema, como serviços de arquivo, serviços de processo, serviços de terminal ou serviços de memória, torna todas as partes pequenas e gerenciáveis. Ademais, como todos os servidores executam como processos de modo usuário e não em modo *kernel*, eles não têm acesso direto ao hardware. Como conseqüência, se ocorrer um *bug* no servidor de arquivos, o serviço de arquivos pode cair, mas isso normalmente não derrubará a máquina inteira.

Outra vantagem do modelo cliente-servidor é sua adaptabilidade para uso em sistemas distribuídos (veja a Figura 1-21). Se um cliente comunica-se com um servidor enviando-lhe mensagens, o cliente não precisa saber se a mensagem é manipulada localmente na própria máquina ou se foi enviada através de uma rede para um servidor em uma máquina remota. No que diz respeito ao cliente, a mesma coisa acontece em ambos os casos: uma requisição foi enviada e uma resposta voltou.

O quadro esboçado acima, de um *kernel* que manipula só o transporte de mensagens de clientes para servidores e vice-versa, não é completamente realista. Algumas funções de sistema operacional (como carregar comandos nos registradores dos dispositivos de E/S físicos) são difíceis, se não impossíveis, de fazer a partir de programas no espaço do usuário. Há duas maneiras de lidar com esse problema.

Figura 1-20 O modelo cliente-servidor.

Uma maneira é ter alguns processos servidores críticos (p. ex., os *drivers* de dispositivo de E/S) executando realmente em modo de *kernel*, com acesso completo a todo o hardware, mas ainda comunicando-se com outros processos, utilizando o mecanismo normal de mensagem.

A outra maneira é construir uma quantidade mínima de **mecanismo** no *kernel*, deixando as decisões **políticas** para os servidores no espaço do usuário. Por exemplo, o *kernel* poderia reconhecer que uma mensagem enviada para determinado endereço significa capturar o conteúdo dessa mensagem e carregá-lo nos registradores de dispositivo de E/S de algum disco, para iniciar a leitura do mesmo. Nesse exemplo, o *kernel* nem mesmo iria inspecionar os bytes na mensagem para ver se são válidos ou significativos: simplesmente os copiaria cegamente para os registradores de dispositivo do disco. (Obviamente, deve-se utilizar algum esquema para limitar essas mensagens a processos autorizados apenas.) A divisão entre mecanismo e política é um conceito importante: ela ocorre repetidamente em sistemas operacionais em diversos contextos.

1.6 VISÃO GERAL DO RESTANTE DESTE LIVRO

Os sistemas operacionais tipicamente têm quatro grandes componentes: gerenciamento de processos, gerenciamento de dispositivos de E/S, gerenciamento de memória e gerenciamento de arquivos. O MINIX também é dividido nessas quatro partes. Os próximos quatro capítulos tratam desses quatro temas, um por capítulo. O Capítulo 6 contêm uma lista de leituras sugeridas e uma bibliografia.

Os capítulos sobre processos, E/S, gerenciamento de memória e sistema de arquivos têm a mesma estrutura geral. Primeiro são expostos os princípios gerais do assunto. Então, é apresentada uma visão geral da área correspondente do MINIX (que também se aplica ao UNIX). Por fim, a implementação do MINIX é discutida em detalhe. A seção de implementação pode ser vista superficialmente ou até pulada sem perda de continuidade para leitores interessados apenas nos princípios dos sistemas operacionais e não no código do MINIX. [Leitores interessados em saber como um sistema operacional real (o MINIX) funciona devem ler todas as seções.]

1.7 RESUMO

Os sistemas operacionais podem ser vistos de dois pontos de vista: gerenciadores de recursos e máquinas estendidas. Na visão de gerenciador de recurso, o trabalho do sistema operacional é gerenciar eficientemente as diferentes partes do sistema. Na visão de máquina estendida, o trabalho do sistema é oferecer aos usuários uma máquina virtual que é mais conveniente para usar do que a máquina real.

Figura 1-21 O modelo cliente-servidor em um sistema distribuído.

Figura 1-20 O modelo cliente-servidor.

Uma maneira é ter alguns processos servidores críticos (p. ex., os *drivers* de dispositivo de E/S) executando realmente em modo de *kernel*, com acesso completo a todo o hardware, mas ainda comunicando-se com outros processos, utilizando o mecanismo normal de mensagem.

A outra maneira é construir uma quantidade mínima de **mecanismo** no *kernel*, deixando as decisões **políticas** para os servidores no espaço do usuário. Por exemplo, o *kernel* poderia reconhecer que uma mensagem enviada para determinado endereço significa capturar o conteúdo dessa mensagem e carregá-lo nos registradores de dispositivo de E/S de algum disco, para iniciar a leitura do mesmo. Nesse exemplo, o *kernel* nem mesmo iria inspecionar os bytes na mensagem para ver se são válidos ou significativos: simplesmente os copiaria cegamente para os registradores de dispositivo do disco. (Obviamente, deve-se utilizar algum esquema para limitar essas mensagens a processos autorizados apenas.) A divisão entre mecanismo e política é um conceito importante: ela ocorre repetidamente em sistemas operacionais em diversos contextos.

1.6 VISÃO GERAL DO RESTANTE DESTE LIVRO

Os sistemas operacionais tipicamente têm quatro grandes componentes: gerenciamento de processos, gerenciamento de dispositivos de E/S, gerenciamento de memória e gerenciamento de arquivos. O MINIX também é dividido nessas quatro partes. Os próximos quatro capítulos tratam desses quatro temas, um por capítulo. O Capítulo 6 contêm uma lista de leituras sugeridas e uma bibliografia.

Os capítulos sobre processos, E/S, gerenciamento de memória e sistema de arquivos têm a mesma estrutura geral. Primeiro são expostos os princípios gerais do assunto. Então, é apresentada uma visão geral da área correspondente do MINIX (que também se aplica ao UNIX). Por fim, a implementação do MINIX é discutida em detalhe. A seção de implementação pode ser vista superficialmente ou até pulada sem perda de continuidade para leitores interessados apenas nos princípios dos sistemas operacionais e não no código do MINIX. [Leitores interessados em saber como um sistema operacional real (o MINIX) funciona devem ler todas as seções.]

1.7 RESUMO

Os sistemas operacionais podem ser vistos de dois pontos de vista: gerenciadores de recursos e máquinas estendidas. Na visão de gerenciador de recurso, o trabalho do sistema operacional é gerenciar eficientemente as diferentes partes do sistema. Na visão de máquina estendida, o trabalho do sistema é oferecer aos usuários uma máquina virtual que é mais conveniente para usar do que a máquina real.

Figura 1-21 O modelo cliente-servidor em um sistema distribuído.

Os sistemas operacionais têm uma longa história, iniciando na época em que substituíram o operador até os modernos sistemas de multiprogramação.

O coração de qualquer sistema operacional é o conjunto de chamadas de sistema que ele pode gerenciar. Essas informam o que o sistema operacional realmente faz. Para o MINIX, essas chamadas podem ser divididas em seis grupos. O primeiro grupo de chamadas de sistema está relacionado com a criação e com o encerramento de processos. O segundo grupo manipula sinais. O terceiro grupo é para ler e para gravar arquivos. Um quarto grupo é para gerenciamento de diretório. O quinto grupo protege informações, e o sexto grupo monitora o tempo.

Os sistemas operacionais podem ser estruturados de vários modos. Os mais comuns são como um sistema monolítico, como uma hierarquia de camadas, como um sistema de máquina virtual e o uso do modelo cliente-servidor.

EXERCÍCIOS

1. Quais são as duas principais funções de um sistema operacional?

2. O que é multiprogramação?

3. O que é *spooling*? Você acredita que os computadores pessoais avançados terão *spooling* como um recurso padrão no futuro?

4. Nos primeiros computadores, cada byte de dados lido ou escrito era diretamente tratado pela CPU (i. e., não havia DMA — acesso direto à memória). Que implicações esse arranjo tem para a multiprogramação?

5. Por que o tempo compartilhado não é comum em computadores de segunda geração?

6. Quais das seguintes instruções devem ser permitidas apenas no modo de *kernel*?
(a) Desativar todas interrupções.
(b) Ler o relógio de hora do dia.
(c) Configurar o relógio de hora do dia.
(d) Mudar o mapeamento da memória.

7. Relacione algumas diferenças entre sistemas operacionais de computadores pessoais e sistemas operacionais de *mainframe*.

8. Um arquivo de MINIX cujo proprietário tem *uid* = 12 e *gid* = 1 tem modo *rwxr-x---*. Outro usuário com *uid* = 6, *gid* = 1 tenta executar o arquivo. O que acontecerá?

9. Em vista do fato de que a mera existência de um superusuário pode levar a todo tipo de problemas de segurança, por que tal conceito existe?

10. O modelo cliente-servidor é popular em sistemas distribuídos. Ele também pode ser utilizado em um sistema de um único computador?

11. Por que a tabela de processos é necessária em um sistema de tempo compartilhado? Ela também é necessária em sistemas de computadores pessoais em que só um processo existe e toma conta da máquina inteira até que se encerre?

12. Qual é a diferença essencial entre um arquivo especial de bloco e um arquivo especial de caractere?

13. No MINIX, se o usuário 2 cria um vínculo para um arquivo possuído pelo usuário 1, e, então, o usuário 1 remove esse arquivo, o que acontece quando o usuário 2 tentar ler o arquivo?

14. Por que a chamada de sistema de CHROOT é limitada ao superusuário? (Sugestão: pense nos problemas de proteção.)

15. Por que o MINIX tem o programa *update* executando em segundo plano o tempo todo?

16. Faz qualquer sentido ignorar o sinal SIGALRM?

17. Escreva um programa (ou uma série de programas) testando todas as chamadas de sistema do MINIX. Para cada chamada, tente vários conjuntos de parâmetros, incluindo alguns incorretos, para ver se eles são detectados.

18. Escreva um *shell* semelhante ao da Figura 1-10 mas contendo código suficiente para realmente funcionar de modo que você, então, possa testá-lo. Você também poderia adicionar alguns recursos tal como redirecionamento de entrada e saída, *pipes* e *jobs* em segundo plano.

2

Processos

Estamos agora prestes a entrar em um estudo detalhado sobre como os sistemas operacionais em geral, e o MINIX em particular, são projetados e construídos. O conceito mais central em qualquer sistema operacional é o de *processo*: uma abstração de um programa em execução. Tudo mais gira em torno desse conceito e é importante que o projetista de sistema operacional (e o estudante) saiba o que é um processo o mais cedo possível.

2.1 INTRODUÇÃO AOS PROCESSOS

Todos os computadores modernos podem fazer várias coisas ao mesmo tempo. Enquanto executa um programa do usuário, um computador também pode estar lendo a partir de um disco e dando saída a texto para uma tela ou impressora. Em um sistema de multiprogramação, a CPU também alterna de um programa para outro, executando cada um por dezenas ou centenas de milissegundos. Enquanto, estritamente falando, em qualquer instante de tempo, a CPU está executando só um programa, no curso de 1 segundo, ela pode funcionar para vários programas, dando aos usuários a ilusão de paralelismo. Às vezes, as pessoas falam de **pseudoparalelismo** querendo referir-se a essa rápida alternância da CPU entre programas, em contraste com o paralelismo verdadeiro em hardware dos sistemas **multiprocessadores** (que têm duas ou mais CPUs compartilhando a mesma memória física). Monitorar múltiplas atividades paralelas é um problema complicado. Assim, com os anos, os projetistas de sistemas operacionais desenvolveram um modelo (processos seqüenciais) que torna o paralelismo mais fácil de tratar. Esse modelo e suas aplicações são o assunto deste capítulo.

2.1.1 Modelo de Processo

Neste modelo, todo o software executável no computador, freqüentemente incluindo o sistema operacional, é organizado em um número de **processos seqüenciais,** ou somente **processos** para simplificar. Um processo é um programa em execução, incluindo os valores atuais do contador de programa, registradores e variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. Na realidade, naturalmente, a CPU alterna de um processo para outro, mas, para entender o sistema, é muito mais fácil pensar em uma coleção de processos que executam em (pseudo)paralelo do que tentar acompanhar como a CPU alterna de um programa para outro. Essa rápida alternância é chamada **multiprogramação,** como vimos no capítulo anterior.

Na Figura 2-1(a), vemos um computador multiprogramado com quatro programas na memória. Na Figura 2-1(b), vemos quatro processos, cada um com seu próprio fluxo de controle (i. e., seu próprio contador de programa) e cada um executando independentemente dos outros. Na Figura 2-1(c), vemos que, a partir de um determinado tempo, todos os processos fizeram progresso, mas em um dado instante só um processo realmente está executando.

Com a CPU alternando entre os processos, a velocidade em que um processo executa sua computação não será uniforme e provavelmente nem mesmo reproduzível se os mesmos processos forem executados novamente. Assim, os processos não devem ser programados com suposições baseadas na coordenação. Considere, por exemplo, um processo de E/S que inicia uma fita de *streamer* para restaurar um backup de arquivos, executa um laço de espera 10.000 vezes para permitir que ela termine o trabalho e,

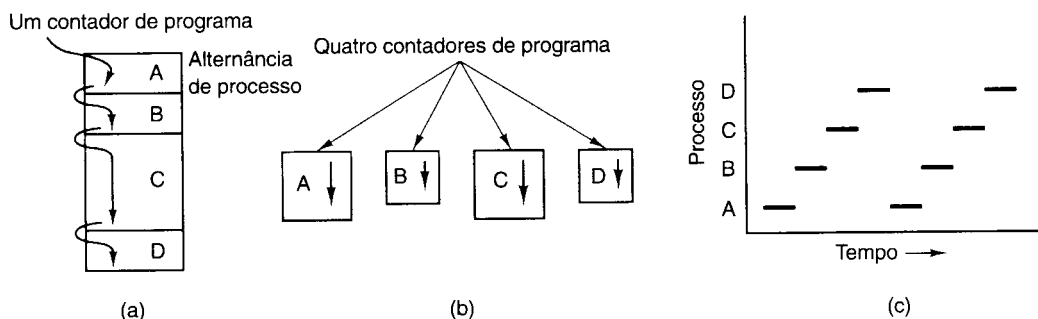



Figura 2-1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos seqüenciais independentes. (c) Só um programa está ativo em qualquer dado instante.

então, dá um comando para ler o primeiro registro. Se a CPU decidir alternar para outro processo durante o laço de espera, o processo da fita pode não executar novamente até que o primeiro registro passe pela cabeça de leitura. Quando um processo tem requisitos de tempo real críticos como esse (i. e., eventos particulares *devem* ocorrer dentro de um número especificado de milissegundos) medidas especiais devem ser tomadas para assegurar que isso ocorra. Normalmente, entretanto, a maioria dos processos não é afetada pela multiprogramação subjacente da CPU, nem pelas velocidades relativas dos diferentes processos.

A diferença entre um processo e um programa é sutil, mas crucial. Uma analogia pode ajudar a tornar mais clara essa questão. Considere um cientista de computador com dotes culinários que está assando um bolo de aniversário para sua filha. Ele tem uma receita de bolo de aniversário e uma cozinha bem-equipada com a entrada necessária: farinha, ovos, açúcar, essência de baunilha, etc. Nessa analogia, a receita é o programa (i. e., um algoritmo expresso em alguma notação conveniente), o cientista de computador é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade que consiste em nosso confeiteiro ler a receita, buscar os ingredientes e cozinhar o bolo.

Agora imagine que o filho do cientista apareça chorando, dizendo que foi picado por uma abelha. O cientista registra onde estava na receita (o estado do processo atual é salvo), procura um livro de pronto-socorro e começa a seguir as orientações nele. Aqui vemos o processador alternando de um processo (cozimento) para um processo de prioridade mais alta (administrar cuidado médico), cada um tendo um programa diferente (receita *versus* livro de pronto-socorro). Quando a picada de abelha foi tratada, o cientista volta ao seu bolo, para continuar a partir do ponto onde ele estava quando abandonou o processo.

A idéia-chave aqui é que um processo é um tipo de atividade. Ele tem um programa, entrada, saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de agendamento sendo utilizado para determinar quando parar de trabalhar em um processo e servir a um diferente.

Hierarquias de Processos

Os sistemas operacionais que suportam o conceito de processo devem fornecer alguma maneira de criar todos os processos necessários. Em sistemas muito simples, ou em sistemas projetados para executar um único aplicativo (p. ex., controlar um dispositivo em tempo real), é possível ter todos os processos que serão necessários alguma vez logo que o sistema inicia. Na maioria dos sistemas, entretanto, é preciso dispor de alguma maneira de criar e de destruir processos conforme necessário durante a operação. No MINIX, os processos são criados pela chamada de sistema FORK, que cria uma cópia idêntica do processo que fez a chamada. O processo-filho também pode executar FORK, então, também é possível obter uma árvore inteira de processos. Em outros sistemas operacionais existem chamadas de sistema para criar um processo, para carregar sua memória e para começar a rodar. Qualquer que seja a natureza exata da chamada de sistema, os processos precisam dispor de uma maneira de criar outros processos. Note que cada processo tem um pai, mas zero, um, dois ou mais filhos.

Como um exemplo simples do modo como as árvores de processo são utilizadas, mostremos o que acontece quando o MINIX é inicializado. Um processo especial, chamado *init* está presente na imagem de inicialização. Quando começa a rodar, ele lê um arquivo, informando quantos terminais existem. Então, ele cria um novo processo por terminal. Esses processos esperam alguém efetuar *login*. Se um *login* é bem-sucedido, o processo de *login* executa um *shell* para receber comandos. Esses comandos podem iniciar mais processos, etc. Assim, todos os processos no sistema inteiro pertencem a uma única árvore, com *init* na raiz.

Estados de um Processo

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, os processos freqüentemente precisam interagir entre si. Um processo pode gerar alguma saída que outro processo utiliza como entrada. No comando de *shell*

```
cat chapter1 chapter2 chapters3 | grep tree
```

o primeiro processo, executando *cat*, dá saída a três arquivos concatenados. O segundo processo, executando *grep*, seleciona todas as linhas que contêm a palavra "tree". Dependendo das velocidades relativas dos dois processos (que dependem da complexidade relativa dos programas e de quanto de tempo de CPU cada um teve), pode acontecer que *grep* esteja pronto para executar, mas não haja nenhuma entrada esperando por ele. Ele deve, então, **bloquear** até que alguma entrada esteja disponível.

Quando um processo bloqueia, ele faz isso porque logicamente ele não pode continuar, em geral, porque está esperando uma entrada que ainda não está disponível. Também é possível que um processo, que está conceitualmente pronto e capaz de executar, seja interrompido porque o sistema operacional decidiu dedicar a CPU a outro processo temporariamente. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (você não pode processar a linha de comando do usuário até que ele a tenha digitado). No segundo caso, é um aspecto técnico do sistema (falta de CPUs suficientes para dar a cada processo seu próprio processador). Na Figura 2-2, vemos um diagrama de estados que mostra os três estados em que um processo pode estar:

1. Executando (realmente utilizando a CPU nesse instante).
2. Pronto (executável; temporariamente parado para permitir que outro processo execute).
3. Bloqueado (incapaz de executar até que algum evento externo aconteça).

Logicamente, os dois primeiros estados são semelhantes. Nos dois casos, o processo está pronto para executar, só que no segundo, não há nenhuma CPU disponível para ele temporariamente. O terceiro estado é diferente dos primeiros dois porque o processo não pode executar, mesmo que a CPU não tenha mais nada a fazer.

Quatro transições são possíveis entre esses três estados, conforme mostrado. A transição 1 ocorre quando um processo descobre que não pode continuar. Em alguns sistemas, o processo deve executar uma chamada de sistema, BLOCK, para entrar no estado bloqueado. Em outros sistemas, incluindo o MINIX, quando um processo lê de um *pipe* ou de um arquivo especial (p. ex., um terminal) e não há nenhuma entrada disponível, o processo é automaticamente bloqueado.

As transições 2 e 3 são causadas pelo agendador de processos, uma parte do sistema operacional, sem que o processo nem mesmo saiba delas. A transição 2 ocorre quando o agendador decide que o processo em execução atuou por tempo suficiente e permite que outro processo tenha algum tempo da CPU. A transição 3 ocorre quando todos os outros processos tiveram sua justa parte e é hora de o primeiro deles receber a CPU para executar novamente. O agendamento, isto é, decidir que processos devem executar quando e por quanto tempo é um assunto importante; iremos examiná-lo mais adiante neste capítulo. Muitos algoritmos foram estudados para tentar equilibrar as demandas de requisição por eficiência para o sistema como um todo e a imparcialidade para os processos individuais.

A transição 4 ocorre quando o evento externo pelo qual um processo está esperando (como a chegada de alguma entrada) acontecer. Se nenhum outro processo está executando nesse instante, a transição 3 será ativada imediatamente e o processo começará a executar. Em vez disso, ele pode ter de esperar em estado *pronto* por alguns instantes até que a CPU esteja disponível.

Usando o modelo de processos, torna-se muito mais fácil pensar no que está ocorrendo dentro do sistema. Alguns processos executam programas que executam comandos digitados por um usuário. Outros processos são parte do sistema e gerenciam tarefas como executar requisições de serviços de arquivo ou gerenciar os detalhes de operação de um disco ou de uma unidade de fita. Quando ocorre uma interrupção de disco, o sistema decide parar de executar o processo atual e executar o processo de disco, que foi bloqueado para esperar essa interrupção. Assim, em vez de pensar nas interrupções, podemos pensar em processos de usuário, em processos de disco, em processos de terminal e assim por diante, que bloqueiam quando estão esperando algo acontecer. Quando o bloco de disco foi lido ou o caractere digitado, o processo em espera é desbloqueado e é elegível para executar novamente.

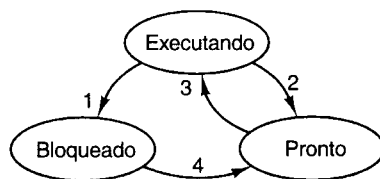
Essa visão dá origem ao modelo mostrado na Figura 2-3. Aqui, o nível mais baixo do sistema operacional é o agendador, com uma variedade de processos nele. Todo o gerenciamento de interrupções e os detalhes sobre como realmente iniciar e parar processos são ocultados do agendador, que é realmente bem pequeno. O restante do siste-

1. O processo bloqueia para entrada
2. O agendador seleciona outro processo
3. O agendador seleciona esse processo
4. A entrada torna-se disponível

Figura 2-2 Um processo pode estar em execução, em estado bloqueado ou pronto. As transições entre esses estados são como mostradas.

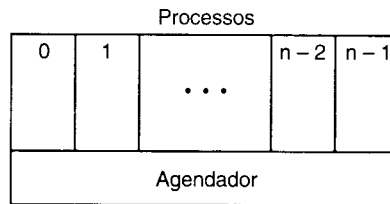



Figura 2-3 A camada mais baixa de um sistema operacional estruturado em processos gerencia interrupções e agendamento. Acima dessa camada, estão os processos seqüenciais.

ma operacional é elegantemente estruturado na forma de processos. O modelo da Figura 2-3 é utilizado no MINIX, com o entendimento de que "agendador" realmente não significa apenas agendamento de processos, mas também o gerenciamento de interrupções e toda a comunicação interprocesso. Mas, como uma primeira abordagem, serve para mostrar a estrutura básica.

2.1.2 Implementação de Processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela (uma matriz de estruturas), chamada **tabela de processos**, com uma entrada por processo. Essa entrada contém as informações sobre o estado do processo, sobre seu contador de programa, sobre o ponteiro da pilha, sobre a alocação de memória, sobre o status de seus arquivos abertos, sobre suas informações de contabilidade e sobre agendamento e tudo mais sobre o processo que deve ser salvo quando o processo alterna de um estado *em execução* para *pronto* a fim de que possa ser reiniciado mais tarde como se nunca tivesse sido interrompido.

No MINIX, o gerenciamento de processos, o gerenciamento de memória e o gerenciamento de arquivo são tratados por módulos separados dentro do sistema, então, a tabela de processos é particionada, com cada módulo mantendo os campos de que precisa. A Figura 2-4 mostra alguns campos mais importantes. Os campos na primeira coluna são os únicos relevantes para este capítulo. As outras duas colunas são fornecidas somente para dar uma idéia das informações que são necessárias em outras partes no sistema.

Agora que vimos a tabela de processos, é possível explicar um pouco mais sobre como a ilusão de múltiplos processos seqüenciais é mantida em uma máquina com uma CPU e muitos dispositivos de E/S. As técnicas a seguir são uma descrição de como o "agendador" da Figura 2-3 funciona no MINIX, mas a maioria dos sistemas operacionais modernos funciona essencialmente da mesma maneira. Associado a cada classe de dispositivo de E/S (p. ex., disquetes, discos rígidos, temporizadores, terminais), existe uma área próxima à parte inferior da memória chamada **vetor de interrupção.** Ele contém o endereço do procedimento de serviço da interrupção. Suponha que o processo de usuário 3 esteja executando quando ocorre uma interrupção de disco. O contador do programa, a palavra de status do programa e possivelmente um ou mais registradores

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores	Ponteiro para segmento de texto	Máscara UMASK
Contador do programa	Ponteiro para segmento de dados	Diretório-raiz
Palavra de status do programa	Ponteiro para segmento bss	Diretório de trabalho
Ponteiro de pilha	Status de saída	Descritores de arquivo
Estado do processo	Status de sinal	Uid efetivo
Tempo em que o processo iniciou	Id do processo	Gid efetivo
Tempo de CPU utilizado	Processo-Pai	Parâmetros de chamada de sistema
Tempo de CPU dos filhos	Grupo do processo	Vários bits de sinalização
Tempo do próximo alarme	Uid real	
Ponteiros de fila de mensagem	Uid efetivo	
Bits de sinal pendente	Gid real	
Id do processo	Gid efetivo	
Vários bits de sinalização	Mapas de bit para sinais	
	Vários bits de flag	

Figura 2-4 Alguns campos da tabela de processos do MINIX.

são enviados para a pilha (atual) pelo hardware de interrupção. O computador, então, salta para o endereço especificado no vetor de interrupção de disco. Isso é tudo o que o hardware faz. Daqui em diante, é com o software.

O procedimento de serviço de interrupção inicia salvando todos os registradores na entrada da tabela de processos para o processo atual. O número do processo atual e um ponteiro para sua entrada são mantidos em variáveis globais de modo que possam ser localizados rapidamente. Então, as informações depositadas pela interrupção são removidas da pilha, e o ponteiro da pilha é modificado para apontar para a pilha temporária utilizada pelo gerenciador de processos. Ações como salvar os registradores e definir o ponteiro da pilha não podem nem mesmo ser expressas em C, então, são executados por uma pequena rotina de linguagem *assembly*. Quando essa rotina termina, ela chama um procedimento em C para fazer o resto do trabalho.

A comunicação interprocesso no MINIX ocorre via mensagens; então, o próximo passo é construir uma mensagem a ser enviada para o processo de disco, que será bloqueado esperando por ele. A mensagem diz que uma interrupção ocorreu para distingui-la de mensagens de processos de usuário solicitando que blocos de disco sejam lidos e coisas semelhantes. O estado do processo de disco agora está mudado de *bloqueado* para *pronto* e o agendador é chamado. No MINIX processos diferentes têm prioridades diferentes, para dar melhor serviço aos manipuladores de dispositivo de E/S do que aos processos de usuário. Se o processo de disco agora é o processo executável de prioridade mais alta, ele será agendado para executar. Se o processo que foi interrompido é igualmente ou mais importante, então, ele será agendado para executar novamente e o processo de disco terá de esperar alguns instantes.

De qualquer modo, o procedimento em C chamado pelo código de interrupção em linguagem *assembly* agora retorna, e o código em linguagem *assembly* carrega os registradores e o mapa de memória para o agora atual processo e inicia sua execução. O gerenciamento e o agendamento de interrupções estão resumidos na Figura 2-5. Vale notar que os detalhes variam ligeiramente de um sistema para outro.

2.1.3 *Threads*

Em um processo tradicional, do tipo que acabamos de estudar, há uma única linha de controle e um único contador de programa em cada processo. Entretanto, em alguns sistemas operacionais modernos, é fornecido suporte para múltiplas linhas de controle dentro de um processo. Essas linhas de controle normalmente são chamadas *threads* ou, ocasionalmente, **processos leves**.

Na Figura 2-6(a) vemos três processos tradicionais. Cada processo tem seu próprio espaço de endereço e uma única linha de controle. Em contraste, na Figura 2-6(b), vemos um único processo com três linhas de controle. Embora em ambos os casos tenhamos três *threads*, na Figura 2-6(a) cada um deles opera em um espaço diferente de endereço, enquanto na Figura 2-6(b) todos os três compartilham o mesmo espaço de endereço.

Como um exemplo de onde múltiplos *threads* podem ser utilizados, considere um processo de servidor de arquivos. Ele recebe requisições para ler e para gravar arquivos e envia de volta os dados requisitados ou aceita os dados atualizados. Para melhorar o desempenho, o servidor mantém um *cache* de arquivos recentemente utilizados em memória, lendo do *cache* e gravando nele quando possível.

Essa situação serve bem para o modelo da Figura 2-6(b). Quando uma requisição entra, ela é passada a um *thread* para processamento. Se esse *thread* bloqueia no meio do caminho para esperar uma transferência de disco, outros *threads* são ainda capazes de executar, assim, o servidor pode continuar processando novas requisições, mesmo quando está ocorrendo E/S de disco. O modelo da Figura 2-6(a) não é adequado, porque é essencial que todos os *threads* do servidor de arquivos acessem o mesmo *cache* e os três *threads* da Figura 2-6(a) não compartilham o mesmo espaço de endereço e, portanto, não podem compartilhar o mesmo *cache* de memória.

Outro exemplo de onde *threads* são úteis está nos navegadores para a World Wide Web, como o Netscape e o Mosaic. Muitas páginas da Web contêm várias pequenas imagens. Para cada imagem em uma página da Web, o navegador deve configurar uma conexão separada com o *site*

1. O hardware empilha o contador de programa, etc.
2. O hardware carrega um novo contador de programa a partir do vetor de interrupção.
3. O procedimento em linguagem assembly salva os registradores.
4. O procedimento em linguagem assembly define uma nova pilha.
5. O serviço interrupção em C executa (em geral, lê e armazena a entrada).
6. O agendador marca a tarefa em espera como pronta.
7. O agendador decide qual processo será executado em seguida.
8. O procedimento em C retorna para o código assembly.
9. O procedimento de linguagem assembly inicia processo atual de novo.

Figura 2-5 O esqueleto do que faz o nível mais baixo do sistema operacional quando ocorre uma interrupção.

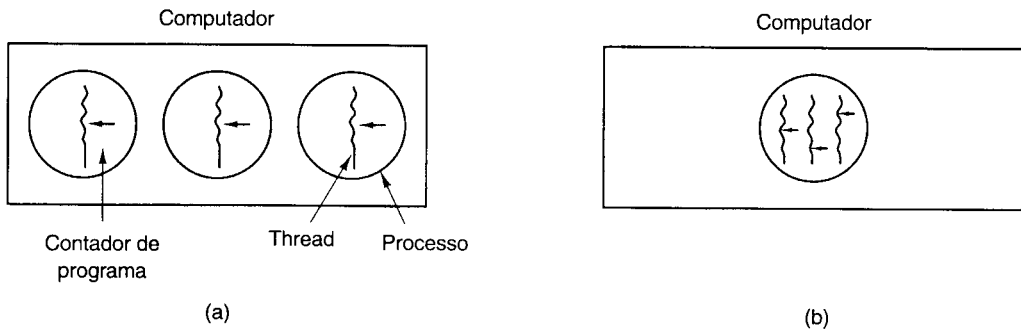



Figura 2-6 (a) Três processos, cada um com um *thread*. (b) Um processo com três *threads*.

da página e requisitar a imagem. Muito tempo é desperdiçado, estabelecendo e liberando todas essas conexões. Por ter múltiplos *threads* dentro do navegador, muitas imagens podem ser solicitadas ao mesmo tempo, acelerando significativamente o desempenho na maioria dos casos, uma vez que com imagens pequenas, o tempo de configuração é o fator limitante, não a velocidade da linha de transmissão.

Quando múltiplos *threads* estão presentes no mesmo espaço de endereço, alguns dos campos da Figura 2-4 não são por processo, mas por *thread*, então uma tabela separada de *thread* é necessária, com uma entrada por *thread*. Entre os itens por *thread*, estão o contador de programa, os registradores e o estado. O contador de programa é necessário porque os *threads*, como os processos, podem ser suspensos e retomados. Os registradores são necessários porque quando os *threads* são suspensos, seus registradores devem ser salvos. Por fim, os *threads*, como os processos, podem estar no estado *em execução*, *pronto* ou *bloqueado*.

Em alguns sistemas, o sistema operacional não está ciente dos *threads*. Em outras palavras, eles são gerenciados inteiramente no espaço do usuário. Quando um *thread* está para bloquear, por exemplo, ele escolhe e inicia seu sucessor antes de parar. Vários pacotes de *threads* no nível do usuário são de uso comum, incluindo os pacotes POSIX ***P-threads*** e Mach ***C-threads***.

Em outros sistemas, o sistema operacional está ciente da existência de múltiplos *threads* por processo, então, quando um *thread* bloqueia, o sistema operacional escolhe o próximo a executar, seja do mesmo processo seja de um diferente. Para fazer o agendamento, o *kernel* deve ter uma tabela de *threads* que lista todos os *threads* no sistema, análoga à tabela de processos.

Embora essas duas alternativas possam parecer equivalentes, elas diferem consideravelmente em desempenho. A comutação de *threads* é muito mais rápida quando o gerenciamento de *threads* é feito no espaço do usuário do que quando uma chamada de *kernel* é necessária. Esse fato é um argumento forte para fazer o gerenciamento de *threads* no espaço do usuário. Por outro lado, quando os *threads* são gerenciados inteiramente no espaço do usuário e um *thread* bloqueia (p. ex., esperando uma E/S ou uma falha de página ser gerenciada), o *kernel* bloqueia o processo inteiro, uma vez que ele nem mesmo está ciente da existência de *threads*. Esse fato é um argumento forte para fazer o gerenciamento de *threads* no *kernel*. Como conseqüência, os dois sistemas estão em uso e também foram propostos vários esquemas híbridos (Anderson *et al.*, 1992).

Independente de os *threads* serem gerenciados pelo *kernel* ou no espaço usuário, eles introduzem diversos problemas que deve ser resolvidos e que mudam o modelo de programação consideravelmente. Para começar considere os efeitos da chamada de sistema FORK. Se o processo-pai tiver múltiplos *threads*, o filho também deve tê-los? Se não, o processo pode não funcionar adequadamente, uma vez que todos eles podem ser essenciais.

Entretanto, se o processo-filho recebe tantos *threads* quanto o pai, o que acontece se um *thread* for bloqueado em uma chamada READ, digamos, do teclado? Dois *threads* agora estão bloqueados no teclado? Quando uma linha é digitada, os dois *threads* obtêm uma cópia dela? Só o pai? Só o filho? O mesmo problema existe com conexões de rede abertas.

Outra classe de problemas está relacionada com o fato de que os *threads* compartilham muitas estruturas de dados. O que acontece se um *thread* fecha um arquivo enquanto outro ainda está lendo esse arquivo? Suponha que um *thread* perceba que há muito pouca memória e comece a alocar mais memória. Então, no meio do caminho, ocorre uma comutação de *threads*, e o novo *thread* também percebe que há pouca memória e também começa a alocar mais memória. A alocação acontece uma ou duas vezes? Em quase todos os sistemas que não foram projetados com *threads* em mente, as bibliotecas (como o procedimento de alocação de memória) não são reentrantes e causarão uma falha se uma segunda chamada for feita enquanto a primeira ainda está ativa.

Outro problema está relacionado com o informe de erros. No UNIX, após uma chamada de sistema, o status da chamada é colocado em uma variável global, *errno*. O que

acontece se um *thread* fizer uma chamada de sistema e, antes de ele ser capaz de ler *errno*, outro *thread* faz uma chamada de sistema, apagando o valor original?

Depois, considere os sinais. Alguns sinais são logicamente específicos ao *thread*, enquanto outros, não. Por exemplo, se um *thread* chama ALARM, faz sentido para o sinal resultante ir para o *thread* que fez a chamada. Quando o *kernel* está ciente dos *threads*, ele normalmente certifica-se de que o *thread* correto obteve o sinal. Quando o *kernel* não está ciente dos *threads*, de alguma maneira o pacote de *threads* deve monitorar os alarmes. Existe uma complicação adicional para *threads* no nível do usuário quando (como no UNIX) um processo só pode ter um alarme pendente por vez e vários *threads* chamam ALARM independentemente.

Outros sinais, como interrupção de teclado, não são específicos ao *thread*. Quem deve capturá-los? Um *thread* específico? Todos os *threads*? Um *thread* recentemente criado? Todas essas soluções têm problemas. Além disso, o que acontece se um *thread* altera os manipuladores de sinal sem informar aos outros *threads*?

Um último problema introduzido por *threads* é o gerenciamento de pilha. Em muitos sistemas, quando ocorre estouro de pilha, o *kernel* apenas fornece mais pilha, automaticamente. Quando um processo tem múltiplos *threads*, ele também deve ter múltiplas pilhas. Se o *kernel* não estiver ciente de todas essas pilhas, ele não poderá aumentá-las automaticamente no caso de falha de pilha. De fato, ele nem mesmo pode saber que uma falha de memória está relacionada com o crescimento da pilha.

Esses problemas certamente não são insuperáveis, mas eles mostram que apenas a introdução de *threads* em um sistema existente sem um reprojeto relativamente substancial do sistema não funcionará absolutamente. A semântica das chamadas de sistema tem de ser redefinida, e as bibliotecas devem ser reescritas, no mínimo. E todas essas coisas devem ser feitas de tal maneira que permaneçam retroativamente compatíveis com programas existentes para o caso limitante de um processo com um só *thread*. Para informações adicionais sobre *threads*, consulte Hauser e colaboradores, 1993, e Marsh e colaboradores, 1991.

2.2 COMUNICAÇÃO INTERPROCESSO

Os processos freqüentemente precisam comunicar-se com outros processos. Por exemplo, em um *pipeline* do *shell*, a saída do primeiro processo deve ser passada para o segundo processo e assim por diante ao longo da linha. Portanto, há uma necessidade de comunicação entre processos, preferivelmente de uma maneira bem-estruturada que não utilize interrupções. Nas seções a seguir, examinaremos algumas questões relacionadas com essa **Comunicação InterProcessos** ou **CIP**.

Bem resumidamente, há três questões aqui. A primeira foi aludida acima: como um processo pode passar as informações para outro. A segunda tem a ver com certificar-se de que dois ou mais processos não interfiram um com outro quando envolvidos em atividades críticas (suponha dois processos tentando agarrar os últimos 100K de memória). A terceira diz respeito ao seqüenciamento adequado quando estão presentes dependências: se o processo *A* produz dados e o processo *B* os imprime, *B* tem de esperar até que *A* tenha produzido alguns dados antes de começar a imprimir. Examinaremos essas três questões agora, começando já na seção seguinte.

2.2.1 Condições de Corrida

Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar algum armazenamento comum que cada um pode ler e gravar. O armazenamento compartilhado pode estar na memória principal ou pode ser um arquivo compartilhado; a localização da memória compartilhada não muda a natureza da comunicação ou os problemas que surgem. Para ver como a comunicação interprocesso funciona na prática, consideraremos um exemplo simples mas comum, um *spooler* de impressão. Quando um processo quer imprimir um arquivo, ele insere o nome de arquivo em um **diretório de spooler** especial. Outro processo, o **servidor de impressão**, verifica periodicamente se há qualquer arquivo a ser impresso e, se houver, ele os imprime e, então, remove seus nomes do diretório.

Imagine que nosso diretório de *spooler* tenha um grande (potencialmente infinito) número de entradas, numeradas como 0, 1, 2, ..., cada uma capaz de armazenar um nome de arquivo. Imagine também que haja duas variáveis compartilhadas: *out*, que aponta para o próximo arquivo a ser impresso, e *in*, que aponta para a próxima entrada livre no diretório. Essas duas variáveis podem ser mantidas em um arquivo de duas palavras disponível para todos os processos. Em um certo instante, as entradas 0 a 3 estão vazias (os arquivos já foram impressos) e as entradas 4 a 6 estão cheias (com os nomes colocados na fila de impressão). Mais ou menos simultaneamente, os processos *A* e *B* decidem que querem colocar um arquivo na fila de impressão. Essa situação é mostrada na Figura 2-7.

Em situações onde a lei de Murphy* é aplicável, pode acontecer o seguinte. O processo *A* lê *in* e armazena o valor, 7, em uma variável local chamada *next_free_slot*. Só que, então, ocorre uma interrupção de relógio, e a CPU decide que o processo *A* executou por tempo suficiente, e, então, alterna para o processo *B*. O processo B também lê *in*, e também recebe um 7, então, ele armazena o nome de seu arquivo no slot 7 e atualiza *in* para que ele seja 8. Então, ele segue adiante e faz outras coisas.

Por fim, o processo *A* executa novamente, iniciando do lugar em que parou. Ele examina *next_free_slot*, encontra um 7 aí e escreve seu nome de arquivo na entrada 7, apagando o nome que o processo *B* acabou de colocar ali.

*N. de R. Se algo pode dar errado, dará.

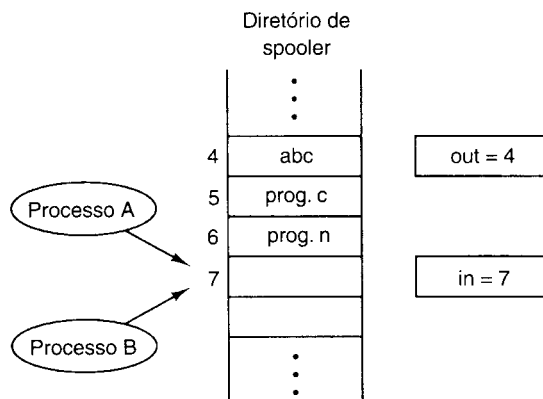



Figura 2-7 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

Então, ele calcula *next_free_slot* + 1, o que dá 8, e define *in* como 8. O diretório do *spooler* está agora internamente consistente, portanto, o servidor de impressão não notará nada de errado, mas o processo *B* nunca realizará qualquer saída. Situações como essa, em que dois ou mais processos estão lendo ou gravando alguns dados compartilhados, e o resultado final depende de quem executa precisamente quando, são chamadas **condições de corrida** (*race conditions*). Depurar programas contendo condições de corrida não é nada divertido. Os resultados da maioria dos testes são bons, mas, de vez em quando, acontece algo estranho e inexplicável.

2.2.2 Seções Críticas

Como evitamos as condições de corrida? A chave para prevenir problemas aqui e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo mais compartilhado é encontrar alguma maneira de proibir que mais de um processo leia e grave os dados compartilhados ao mesmo tempo. Dito em outras palavras, precisamos de uma **exclusão mútua** — uma maneira de certificarmo-nos de que se um processo está utilizando um arquivo ou variável compartilhado, os outros processos serão impedidos de fazer a mesma coisa. A dificuldade acima ocorria porque o processo *B* começava utilizando uma das variáveis compartilhadas antes de o processo *A* ter acabado de trabalhar com ela. A escolha das operações primitivas apropriadas para obter a exclusão mútua é uma questão de projeto importante em qualquer sistema operacional e um assunto que examinaremos detalhadamente nas seções a seguir.

O problema de evitar as condições de corrida também pode ser formulado de uma maneira abstrata. Parte do tempo, um processo fica ocupado fazendo computações internas e outras coisas que não conduzem a condições de corrida. Entretanto, às vezes, um processo pode estar acessando memória compartilhada ou arquivos compartilhados, ou fazer outras coisas críticas que podem levar à condições de corrida. Essa parte do programa em que a memória compartilhada é acessada é chamada **região crítica** ou **seção crítica.** Se pudéssemos organizar os problemas de tal modo que nenhum dos dois processos jamais estivesse em suas regiões críticas ao mesmo tempo, poderíamos evitar as condições de corrida.

Embora esse requisito evite as condições de corrida, isso não é suficiente para ter processos paralelos que cooperam correta e efetivamente, utilizando dados compartilhados. Precisamos sustentar quatro condições para ter uma boa solução:

1. Nenhum dos dois processos pode estar simultaneamente dentro de suas regiões críticas.
2. Nenhuma suposição pode ser feita sobre as velocidades ou sobre o número de CPUs.
3. Nenhum processo que executa fora de sua região crítica pode bloquear outro processo.
4. Nenhum processo deve ter de esperar eternamente para entrar em sua região crítica.

2.2.3 Exclusão Mútua com Espera Ativa

Nesta seção, examinaremos várias propostas para obter exclusão mútua, de modo que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo entrará em *sua* região crítica para causar problemas.

Desativando as Interrupções

A solução mais simples é fazer cada processo desativar todas as interrupções imediatamente depois de ele entrar em sua região crítica e reativá-las imediatamente depois de ele sair dela. Com as interrupções desativadas, nenhuma interrupção de relógio pode ocorrer. A CPU só alterna

de um processo para outro como resultado de interrupções de relógio ou de outras interrupções; no final das contas, com as interrupções desligadas a CPU não alternará de um processo para outro. Assim, uma vez que um processo desativou as interrupções, ele pode examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo intervirá.

Essa abordagem é geralmente pouco atraente porque não é aconselhável dar o poder de desativar interrupções a processos de usuário. Suponha que um deles faça isso e nunca mais as ative novamente? Isso poderia ser o fim do sistema. Além disso, se o sistema é multiprocessado, com duas ou mais CPUs, desativar interrupções afeta só a CPU que executou a instrução de desativamento. As outras continuarão executando e podem acessar a memória compartilhada.

Por outro lado, freqüentemente é conveniente para o próprio *kernel* desativar interrupções pelo tempo de algumas instruções enquanto ele está atualizando variáveis ou listas. Se uma interrupção ocorreu enquanto a lista de processos prontos, por exemplo, estava em um estado inconsistente, poderiam ocorrer condições de corrida. A conclusão é: desativar interrupções é freqüentemente uma técnica útil dentro do sistema operacional em si, mas não é apropriada como um mecanismo geral de exclusão mútua para processos de usuário.

Variáveis de Bloqueio

Como uma segunda tentativa, vamos procurar uma solução de software. Considere ter uma variável única compartilhada (bloqueio) inicialmente como 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa o bloqueio. Se o bloqueio for 0, o processo o define como 1 e entra na região crítica. Se o bloqueio já for 1, o processo apenas espera até ele tornar-se 0. Assim, 0 significa que nenhum processo está em sua região crítica e 1 significa que algum processo está em sua região crítica.

Infelizmente, essa idéia contém exatamente o mesmo defeito fatal que vimos no diretório do *spooler*. Suponha que o processo leia o bloqueio e veja que ele é 0. Antes de poder definir o bloqueio como 1, outro processo é agendado, executa e define o bloqueio como 1. Quando o primeiro processo executa novamente, ele também definirá o bloqueio como 1, e os dois processos estarão em suas regiões críticas ao mesmo tempo.

Agora você pode pensar que poderíamos evitar esse problema lendo primeiro o valor de bloqueio e, então, verificando-o novamente, imediatamente antes de armazenar nele, mas isso na verdade não ajuda. A condição de corrida agora ocorre se o segundo processo modifica o bloqueio imediatamente depois de o primeiro processo acabar de fazer sua segunda verificação.

Alternância Estrita

Uma terceira abordagem para o problema de exclusão mútua é mostrada na Figura 2-8. Esse fragmento de programa, como quase todos os outros neste livro, é escrito em C. A linguagem C foi escolhida aqui, porque os sistemas operacionais reais comumente são escritos em C (ou ocasionalmente em C++), mas quase nunca em linguagens como Modula 2 ou Pascal.

Na Figura 2-8, a variável de número inteiro *turn*, inicialmente 0, monitora aquele de quem é a vez (*turn*) de entrar na região crítica e examinar ou atualizar a memória compartilhada. Inicialmente, o processo 0 inspeciona *turn*, descobre que ele é 0 e entra na sua região crítica. O processo 1 também descobre que ele é 0 e, portanto, entra em um laço estrito testando *turn* continuamente para ver quando ele se torna 1. Testar continuamente uma variável até que algum valor apareça é chamado **espera ativa.** Normalmente deve ser evitado, uma vez que desperdiça tempo de CPU. Só quando há uma expectativa razoável de que a espera seja curta é que a espera ativa é utilizada.

Quando o processo 0 sai da região crítica, ele define *turn* como 1, permitindo que o processo 1 entre em sua região crítica. Suponha que o processo 1 termine de trabalhar em sua região crítica rapidamente, então, ambos os processos estão em suas regiões não-críticas, com *turn* configurado como 0. Agora o processo 0 executa seu laço inteiro rapidamente, voltando para sua região não-crítica com *turn* configurado como 1. Nesse ponto, o processo 1 acaba de trabalhar na sua região não-crítica e volta ao topo do seu laço. Infelizmente, ele não tem permissão para entrar na sua região crítica agora, porque *turn* está configurado como 1 e o processo 1 está ocupado com sua região não-crítica. Colocado de maneira diferente, a utilização de turnos (*turn*) não é uma boa idéia quando um dos processos é muito mais lento do que o outro.

Essa situação viola a condição 3 estabelecida anteriormente: o processo 1 está sendo bloqueado por um processo

```
while (TRUE) {
  while (turn != 0) /* espera */;
  critical_region();
  turn = 1;
  noncritical_region();
}
```

(a)

```
while (TRUE) {
  while (turn != 1 ) /* espera */;
  critical_region();
  turn = 0;
  noncritical_region();
}
```

(b)

Figura 2-8 Uma solução proposta para o problema da região crítica.

que não está em sua região crítica. Voltando ao diretório de *spooler* já discutido, se agora associássemos a região crítica com leitura e com gravação do diretório de *spooler*. o processo 0 não teria permissão para imprimir outro arquivo porque o processo 1 estaria fazendo outra coisa.

De fato, essa solução requer que os dois processos alternem estritamente sua entrada nas suas regiões críticas, por exemplo, em arquivos de *spool*. Nenhum deles teria permissão para fazer dois *spools* em fila. Embora esse algoritmo realmente evite todas as condições de corridas, na realidade, ele não é um candidato sério como uma solução porque viola a condição 3.

A Solução de Peterson

Combinando a idéia de turnos com a idéia de variáveis de bloqueio e variáveis de aviso, o matemático holandês T. Dekker foi o primeiro a projetar uma solução de software para o problema da exclusão mútua que não requer alternância estrita. Para uma discussão sobre o algoritmo de Dekker, consulte (Dijkstra, 1965).

Em 1981, G.L. Peterson descobriu um modo muito mais simples de obter a exclusão mútua, tornando obsoleta assim a solução de Dekker. O algoritmo de Peterson é mostrado na Figura 2-9. Esse algoritmo consiste em dois procedimentos escritos em ANSI C, o que significa que protótipos de função devem ser fornecidos para todas as funções definidas e utilizadas. Entretanto, para poupar espaço, não mostraremos os protótipos neste exemplo nem nos subseqüentes.

Antes de utilizar as variáveis compartilhadas (i. e., antes de entrar em sua região crítica), cada processo chama *enter_region* com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada causará espera, se necessário, até que seja seguro entrar. Depois que terminou de trabalhar com as variáveis compartilhadas, o processo chama *leave_region* para indicar que terminou e permitir que o outro processo entre, se ele, então, quiser.

Deixe-nos ver como essa solução funciona. Inicialmente nenhum processo está em sua região crítica. Agora o processo 0 chama *enter_region*. Ele indica seu interesse, configurando seu elemento da matriz e configura *turn* como 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 agora chamar *enter_region*, ele ficará parado aí até que *interested[0]* torne-se *FALSE*, evento que só acontece quando o processo 0 chama *leave_region* para sair da região crítica.

Agora, considere o caso em que os dois processos chamam *enter_region* quase simultaneamente. Ambos armazenarão seu número de processo em *turn*. Qualquer que seja o armazenamento feito por último, é este que conta; o primeiro é perdido. Suponha que o processo 1 armazene por último, assim *turn* é 1. Quando ambos os processos chegam na declaração *while*, o processo 0 a executa zero vezes e entra em sua região crítica. O processo 1 entra em laço e não entra em sua região crítica.

A Instrução TSL

Agora vejamos uma proposta que requer uma pequena ajuda do hardware. Muitos computadores, especialmente aqueles projetado com múltiplos processadores em mente, têm uma instrução TEST AND SET LOCK (TSL — testa e configura o bloqueio) que funciona como segue. Ela lê o conteúdo da palavra de memória em um registrador e, então, armazena um valor diferente de zero nesse endereço de memória. As operações de leitura e de armazenamento da palavra são garantidas como sendo indivisíveis — nenhum outro processador pode acessar a palavra de memória até que a instrução tenha acabado. A CPU que executa a ins-

```
#define FALSE   0
#define TRUE    1
#define N       2                                       /* número de processos */

int turn;                                               /* de quem é a vez (turn)? */
int interested[N];                                      /* todo os valores inicialmente 0 (FALSE) */

void enter_region(int process);                         /* o process é 0 ou 1 */
{
 int other;                                             /* número dos outros processos */

 other = 1 – process;                                   /* o oposto do processo */
 interested[process] = TRUE;                            /* mostra que você está interessado */
 turn = process;                                        /* define o sinalizador */
 while (turn == process && interested[other] == TRUE)   /* declaração nula */;
}

void leave region(int process)                          /* processo: quem está saindo*/
{
 interested[process] = FALSE;                           /* indica saída da região crítica */
}
```

Figura 2-9 A solução de Peterson para obter exclusão mútua.

trução TSL bloqueia o barramento de memória, proibindo outras CPUs de acessar a memória até que ela tenha terminado.

Para utilizar a instrução TSL, utilizaremos uma variável compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando *lock* é 0, qualquer processo pode configurá-la como 1 utilizando a instrução TSL e, então, ler ou gravar a memória compartilhada. Quando termina, o processo configura *lock* de volta para 0 utilizando uma instrução MOVE comum.

Como essa instrução pode ser utilizada para prevenir que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Figura 2-10. Essa figura mostra uma sub-rotina de quatro instruções em uma fictícia (mas típica) linguagem *assembly*. A primeira instrução copia o valor antigo de *lock* para o registrador e, então, configura *lock* como 1. Então, o valor antigo é comparado com 0. Se for diferente de zero, o bloqueio já foi definido e, então, o programa volta para o começo e testa novamente. Cedo ou tarde, ele se tornará 0 (quando o processo atualmente em sua região crítica terminar de trabalhar nela) e a sub-rotina retorna com o bloqueio configurado. Limpar o bloqueio é simples. O programa simplesmente armazena um 0 em *lock*. Nenhuma instrução especial é necessária.

Uma solução para o problema da região crítica agora é simples. Antes de entrar em sua região crítica, um processo chama *enter_region*, que faz a espera ativa até que o bloqueio esteja livre; então, ele adquire o bloqueio e retorna. Depois da região crítica, o processo chama *leave_region*, que armazena um 0 em *lock*. Como ocorre em todas as soluções baseadas em regiões críticas, os processos devem chamar *enter_region* e *leave_region* nos momentos certos para o método funcionar. Se um processo trapacear, a exclusão mútua falhará.

2.2.4 *Sleep* e *Wakeup*

Tanto a solução de Peterson como a solução que utiliza TSL são corretas, mas ambas têm o defeito de necessitar da espera ativa. Essencialmente, o que essas soluções fazem é isto: quando um processo quer entrar em sua região crítica, ele verifica se a entrada é permitida. Se não for, o processo apenas espera em um laço estrito até que seja.

Essa abordagem não apenas desperdiça tempo de CPU como também pode ter efeitos inesperados. Considere um computador com dois processos, *H*, com alta prioridade, e *L*, com baixa prioridade. As regras de agendamento são tais que *H* é executado sempre que está em estado pronto. Em um certo momento, com *L* em sua região crítica, *H* torna-se pronto para executar (p. ex., uma operação de E/S completa-se). *H* agora começa a espera ativa, mas como *L* nunca é agendado quando *H* está executando, *L* nunca tem a chance de sair da sua região crítica, então *H* fica em um laço eterno. Essa situação é, às vezes, referida como **problema da inversão de prioridade.**

Agora vejamos algumas primitivas de comunicação interprocesso que bloqueiam em vez de desperdiçar tempo de CPU quando a eles não é permitido entrar em suas regiões críticas. Uma das mais simples é o par SLEEP e WAKEUP. SLEEP é uma chamada de sistema que causa o bloqueio do processo que fez a chamada, isto é, ele é suspenso até que outro processo o acorde. A chamada WAKEUP tem um parâmetro, o processo a ser acordado. Alternativamente, SLEEP e WAKEUP têm um parâmetro, um endereço de memória utilizado para coincidir os SLEEPs com os WAKEUPs.

O Problema dos Produtores e Consumidores

Como um exemplo de como essas primitivas podem ser utilizadas, consideremos o **problema dos produtores e dos consumidores** (também conhecido como problema do **buffer associado**). Dois processos compartilham um buffer de tamanho fixo. Um deles, o produtor, coloca as informações em um buffer e o outro, o consumidor, pega-as. (Também é possível generalizar o problema para ter *m* produtores e *n* consumidores, mas consideraremos apenas o caso de um produtor e de um consumidor porque essa suposição simplifica as soluções).

O problema surge quando o produtor quer colocar um novo item no buffer, mas este último já está cheio. A solução é o produtor ir dormir para ser acordado quando o consumidor remover um ou mais itens. De maneira semelhante, se o consumidor quiser remover um item do buffer e ver que o buffer está vazio, ele adormecerá até que o produtor coloque algo no buffer e o acorde.

Essa abordagem parece suficientemente simples, mas conduz ao mesmo tipo de condição de corrida que vimos anteriormente com o diretório de *spooler*. Para monitorar o número de itens no buffer, precisaremos de uma variá-

```
enter_region:
        tsl register,lock        | copia lock para o registrador e o configura como 1
        cmp register,#0          | lock era zero?
        jne enter region         | se não era zero, o bloqueio estava configurado, então inicia um laço
        ret                      | retorna para aquele que fez a chamada; entrada na região crítica


leave_region:
        move lock,#0             | armazena um 0 no bloqueio (lock)
        ret                      | retorna para aquele que fez a chamada
```

Figura 2-10 Configurando e limpando bloqueios utilizando TSL.

vel, *count*. Se o número máximo de itens que o buffer pode armazenar for *N*, o código do produtor primeiro testará se *count* é *N*. Se for, o produtor adormecerá; se não, o produtor adicionará um item e aumentará *count*.

O código para o consumidor é semelhante: primeiro testa *count* para ver se é 0. Se for, adormecerá; se for diferente de zero, removerá um item e diminuirá o contador. Cada um dos processos também testa para ver se o outro deveria estar dormindo e, se não, ele o acorda. Os códigos para o produtor e para o consumidor são mostrados na Figura 2-11.

Para expressar chamadas de sistema como SLEEP e WAKEUP em C, nós as mostraremos como chamadas para rotinas de biblioteca. Elas não são parte da biblioteca C padrão, mas presumivelmente estariam disponíveis em qualquer sistema que realmente tivesse essas chamadas de sistema. Os procedimentos *enter_item* e *remove_item*, que não são mostrados, gerenciam o trabalho de colocar itens no buffer e remover itens do mesmo.

Agora voltemos à condição de corrida. Ela pode ocorrer porque o acesso a *count* é irrestrito. A seguinte situação possivelmente poderia ocorrer. O buffer está vazio e o consumidor terminou de ler *count* para ver se ele é 0. Nesse instante, o agendador decide parar de executar o consumidor temporariamente e começar a executar o produtor. O produtor insere um item no buffer, aumentando *count e* avisa que ele agora é 1. Deduzindo que *count* era apenas 0, e assim o consumidor deveria estar dormindo, o produtor chama *wakeup* para acordar o consumidor.

Infelizmente o consumidor ainda não está logicamente adormecido, então, o sinal para acordar é perdido. Quando o consumidor executar da próxima vez, ele testará o valor de *count* anteriormente lido, verificará que ele é 0, e, então, irá dormir. Cedo ou tarde, o produtor encherá o buffer e também irá dormir. Ambos dormirão eternamente.

A essência do problema aqui é que um sinal para acordar enviado para um processo que não está dormindo (ainda) é perdido. Se ele não fosse perdido, tudo funcionaria. Uma rápida correção é modificar as regras, adicionando um **bit de espera por despertar** ao quadro geral. Quando um sinal para acordar é enviado para um processo que ainda está acordado, esse bit é ativado. Mais tarde, quando o processo tenta ir dormir, se o bit de espera por despertar estiver ativado, ele será desativado, mas o processo permanecerá acordado. O bit de espera por despertar é um banco para sinais de acordar.

Embora o bit de espera por despertar "salve a pátria" nesse exemplo simples, é fácil construir exemplos com três ou mais processos em que um bit de espera por despertar é insuficiente. Poderíamos fazer outro remendo e adicionar um segundo bit de espera por despertar ou talvez 8 ou 32 deles, mas, em princípio, o problema ainda está aí.

2.2.5 Semáforos

Esta era a situação em 1965, quando E. W. Dijkstra (1965) sugeriu utilizar uma variável inteira para contar o número de *wakeups* salvos para uso futuro. Em sua proposta, um novo tipo de variável chamado **semáforo** foi introduzido. Um semáforo poderia ter o valor 0, indicando que nenhum *wakeup* foi salvo ou algum valor positivo se um ou mais *wakeups* estivessem pendentes.

Dijkstra propôs ter duas operações DOWN e UP (generalizações de SLEEP e WAKEUP, respectivamente). A operação

```
#define N 100                                   /* número de entradas no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
 while (TRUE) {                                 /* repete eternamente */
        produce_item();                         /* gera o item seguinte */
        if (count== N) sleep();                 /* se o buffer estiver cheio, vai dormir */
        enter_ item();                          /* coloca item no buffer */
        count = count + 1;                      /* aumenta a contagem de items no buffer */
        if (count == 1) wakeup(consumer);       /* o buffer estava vazio? */
 }
}


void consumer(void)
{
 while (TRUE) {                                 /* repete eternamente */
        if (count == 0) sleep();                /* se o buffer estiver vazio, vai dormir */
        remove_item();                          /* remove item do buffer */
        count = count – 1;                      /* diminiu a contagem de itens no buffer */
        if (count == N - 1) wakenp (producer);  /* o buffer estava cheio? */
        consume_item();                         /* imprime o item */
 }
}
```

Figura 2-11 O problema dos produtores e consumidores com uma condição de corrida fatal.

DOWN em um semáforo verifica se o valor é maior que 0. Se for, ele diminui o valor (i. e., utiliza um *wakeup* armazenado) e simplesmente continua. Se o valor for 0, o processo é colocado para dormir sem completar o DOWN, por enquanto. Verificar o valor, alterá-lo e, possivelmente, ir dormir é tudo feito como uma única e indivisível **ação atômica.** É garantido que uma vez que uma operação de semáforo iniciou, nenhum outro processo acesse o semáforo até que a operação tenha-se se completado ou tenha sido bloqueada. Essa atomicidade é absolutamente essencial para resolver problemas de sincronização e para evitar condições de corrida.

A operação UP incrementa o valor do semáforo endereçado. Se um ou mais processos estavam dormindo nesse semáforo, incapazes de completar uma operação DOWN anterior, um deles é escolhido pelo sistema (p. ex., aleatoriamente) e autorizado a completar seu DOWN. Assim, depois de um UP em um semáforo com processos adormecidos nele, o semáforo ainda será 0, mas haverá um processo a menos dormindo nele. A operação de incrementar o semáforo e de acordar um processo também é indivisível. Nenhum processo bloqueia, utilizando um UP, assim como nenhum processo bloqueia, fazendo um WAKEUP no modelo anterior.

A propósito do paper original de Dijkstra, ele utilizou os nomes P e V em vez de DOWN e UP, respectivamente, mas como aqueles não têm nenhum importância mnemônica para as pessoas que não falam holandês (e importância só marginal para aqueles que falam) utilizaremos os termos DOWN e UP no lugar deles. Esses foram introduzidos pela primeira vez no Algol 68.

Resolvendo o Problema dos Produtores e dos Consumidores Utilizando Semáforos

Os semáforos resolvem o problema do *wakeup* perdido como mostrado na Figura 2-12. É essencial que eles sejam implementados de uma maneira indivisível. A maneira normal é implementar UP e DOWN como chamadas de sistema com o sistema operacional brevemente desativando todas as interrupções, enquanto está testando o semáforo, atualizando-o e colocando o processo para dormir, se necessário. Como todas essas ações precisam de apenas algumas instruções, não há nenhum prejuízo em desativar as

```
#define N 100                          /* número de entradas no buffer */
typedef int semaphore;                 /* semáforos são um tipo especial de int */
semaphore mutex = 1;                   /* controle de acesso à região crítica */
semaphore empty = N;                   /* conta as entradas livres do buffer */
semaphore full = 0;                    /* conta as entradas utiizadas do buffer */

void producer(void)
{
 int item;

 while (TRUE) {                        /* TRUE é a constante 1 */
        produce_item(&item);           /* gera algo para colocar no buffer */
        down(&empty);                  /* diminui a contagem de entradas livres */
        down(&mutex);                  /* entra na região crítica */
        enter_item(item);              /* coloca um novo item no buffer */
        up(&mutex);                    /* sai da região crítica */
        up(&full);                     /* aumenta a contagem de entradas utilizadas */

 }
}

void consumer(void)
{
 int item;

 while (TRUE) {                        /* laço infinito */
        down (&full);                  /* diminui a contagem de entradas utilizadas */
        down(&mutex);                  /* entra na região crítica */
        remove_item(&item);            /* remove item do buffer */
        up(&mutex);                    /* sai de região crítica */
        up(&empty);                    /* aumenta a contagem de entradas livres
        consume_item(item);            /* faz alguma coisa com o item */
 }
}
```

Figura 2-12 O problema dos produtores e dos consumidores utilizando semáforos.

interrupções. Se múltiplas CPUs estiverem sendo utilizadas, cada semáforo deve ser protegido por uma variável de bloqueio com a instrução de TSL utilizada para certificar que só uma CPU por vez examine o semáforo. Certifique-se de que você entendeu que utilizar TSL para evitar que várias CPUs acessem o semáforo ao mesmo tempo é bem diferente da espera ativa pelo produtor ou pelo consumidor, esperando o outro esvaziar ou encher o buffer. A operação de semáforo só leva alguns microssegundos enquanto o produtor ou o consumidor podem demorar um tempo arbitrariamente longo.

Essa solução utiliza três semáforos: um chamado *full* para contar o número de entradas que estão ocupadas, um chamado *empty* para contar o número de entradas que estão livres e um chamado *mutex* para assegurar que o produtor e o consumidor não acessem o buffer ao mesmo tempo. Inicialmente *full* é 0, *empty* é igual ao número de entradas no buffer e *mutex* também é 1. Os semáforos que são inicializados como 1 e utilizados por dois ou mais processos para assegurar que só um deles possa entrar em sua região crítica ao mesmo tempo são chamadas **semáforos binários.** Se cada processo faz um DOWN imediatamente antes de entrar em sua região crítica e um UP somente depois de sair dela, a exclusão mútua é garantida.

Agora que temos uma boa primitiva de comunicação interprocesso à nossa disposição, voltemos a ver as seqüência de interrupções da Figura 2-5. Em um sistema que utiliza semáforos, a maneira natural de ocultar interrupções é ter um semáforo inicialmente configurado como 0, associado com cada dispositivo de E/S. Imediatamente depois de iniciar um dispositivo de E/S, o processo gerenciador faz um DOWN no semáforo associado, bloqueando, assim, de imediato. Quando chega a interrupção, o gerenciador de interrupções faz um UP no semáforo associado, o qual torna o processo relevante pronto para executar novamente. Nesse modelo, o passo 6 na Figura 2-5 consiste em fazer um UP no semáforo do dispositivo, de modo que no passo 7, o agendador será capaz de executar o gerenciador de dispositivo. Naturalmente, se agora vários processos estão prontos, o agendador pode escolher executar um processo ainda mais importante em seguida. Veremos como o agendamento é feito mais adiante neste capítulo.

No exemplo da Figura 2-12, utilizamos os semáforos, na realidade, de duas maneiras diferentes. Essa diferença é suficientemente importante para merecer ser explicitada. O semáforo *mutex* é utilizado para exclusão mútua. Ele é projetado para garantir que só um processo por vez estará lendo ou gravando o buffer e as variáveis associadas. Essa exclusão mútua é requerida para evitar um caos.

A outra utilização de semáforos é para **sincronização.** Os semáforos *full* e *empty* são necessários para garantir que certas seqüências de eventos ocorram ou não. Nesse caso, eles asseguram que o produtor pára de executar quando o buffer está cheio, e o consumidor pára de executar quando o buffer está vazio. Esse uso é diferente da exclusão mútua.

Embora a idéia dos semáforos exista há mais de um quarto de século, muitos estudiosos ainda estão fazendo pesquisa sobre sua utilização. Como um exemplo, consulte (Tai e Carver, 1996).

2.2.6 Monitores

Com os semáforos, a comunicação interprocesso parece fácil, certo? Esqueça. Examine de perto a ordem dos DOWNs antes de inserir ou de remover itens do buffer na Figura 2-12. Suponha que os dois DOWNs no código do produtor foram invertidos na sua ordem, então, *mutex* foi diminuído antes de *empty* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor bloquearia com *mutex* configurado como 0. Portanto, da próxima vez que o consumidor tentasse acessar o buffer, ele faria um DOWN em *mutex*, agora 0, e também bloquearia. Os dois processos permaneceriam bloqueados eternamente e mais nenhum trabalho seria feito. Essa situação infeliz é chamada **impasse** (*deadlock*). Estudaremos impasses detalhadamente no Capítulo 3.

Esse problema é indicado para mostrar quanto cuidado deve-se ter ao utilizar semáforos. Um erro sutil e tudo desaba. É como programação em linguagem *assembly*, só que pior, porque os erros são condições de corrida, de impasses e de outras formas de comportamento imprevisíveis e irreproduzíveis.

Para tornar mais fácil escrever programas corretos, Hoare (1974) e Brinch Hansen (1975) propuseram uma primitiva de sincronização de nível mais alto chamada **monitor.** Suas propostas diferiam ligeiramente como descrito adiante. Um monitor é uma coleção de variáveis, de procedimentos e de estruturas de dados que são agrupados em um tipo especial de módulo ou de pacote. Os processos podem chamar os procedimentos em um monitor sempre que quiserem, mas eles não podem acessar diretamente as estruturas de dados internas do monitor a partir de procedimentos declarados fora do monitor. A Figura 2-13 ilustra um monitor escrito em uma linguagem imaginária, semelhante ao Pascal.

Os monitores têm uma propriedade importante que os torna úteis para obter a exclusão mútua: só um processo pode estar ativo em um monitor em qualquer instante. Os monitores são uma construção de linguagem de programação, de modo que o compilador sabe que eles são especiais e pode gerenciar chamadas para procedimentos do monitor diferentemente de outras chamadas de procedimento. Em geral, quando um processo chama um procedimento do monitor, as primeiras poucas instruções do procedimento verificarão se qualquer outro processo está atualmente ativo dentro do monitor. Se estiver, o processo de chamada será suspenso até que o outro processo tenha deixado o monitor. Se nenhum outro processo estiver utilizando o monitor, o processo de chamada pode entrar.

Cabe ao compilador implementar a exclusão mútua em entradas de monitor, mas uma maneira comum é utilizar

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

  procedure consumer(x);
  .
  .
  .
  end;
end monitor;
```

Figura 2-13 Um monitor.

um semáforo binário. Como é o compilador, e não o programador, que está organizando a exclusão mútua, é muito menos provável que algo dê errado. De qualquer modo, a pessoa que escreve o monitor não precisa estar ciente de como o compilador organiza a exclusão mútua. É suficiente saber que transformando todas as regiões críticas em procedimentos de monitor, nunca dois processos executarão suas regiões críticas ao mesmo tempo.

Embora os monitores ofereçam uma maneira fácil de obter exclusão mútua, como vimos anteriormente, isso não é suficiente. Também precisamos de uma maneira de bloquear os processos quando eles não podem prosseguir. No problema do produtor-consumidor, é muito fácil colocar todos os testes para buffer cheio e para buffer vazio em procedimentos de monitor, mas como o produtor deve bloquear quando encontra o buffer cheio?

A solução reside na introdução de **variáveis de condição,** junto com duas operações sobre elas, WAIT e SIGNAL. Quando um procedimento de monitor descobre que não pode continuar (p. ex., o produtor encontra o buffer cheio), ele faz um WAIT em alguma variável de condição, digamos, *full*. Essa ação causa o bloqueio do processo de chamada. Ela também permite que outro processo, que anteriormente tinha sido proibido de entrar no monitor, entre.

Nesse outro processo, por exemplo, o consumidor pode acordar seu parceiro adormecido, fazendo um SIGNAL na variável de condição que seu parceiro está esperando. Para evitar ter dois processos ativos no monitor, ao mesmo tempo, precisamos de uma regra informando o que acontece após um SIGNAL. Hoare propôs deixar o processo recentemente acordado executar, suspendendo os outros. Brinch Hansen propôs refinar o problema requerendo que um processo que faz um SIGNAL saia do monitor imediatamente. Em outras palavras, uma declaração SIGNAL pode aparecer apenas como a declaração final em um procedimento de monitor. Utilizaremos a proposta de Brinch Hansen por ser conceitualmente mais simples e, também, mais fácil de implementar. Se um SIGNAL é feito em uma variável de condição em que vários processos estão esperando, só um deles, determinado pelo agendador do sistema, é reanimado.

As variáveis de condição não são contadores. Elas não acumulam sinais para uso futuro do modo como os semáforos fazem. Assim, se uma variável de condição é sinalizada com ninguém esperando nela, o sinal é perdido. O WAIT deve vir antes do SIGNAL. Essa regra torna a implementação muito mais simples. Na prática, não é um problema porque é fácil acompanhar o estado de cada processo com variáveis, se for necessário. Qualquer processo que pode fazer um SIGNAL verá que essa operação não é necessária examinando as variáveis.

Um esqueleto do problema dos produtores e dos consumidores com monitores é dado na Figura 2-14 em código-fonte parecido com o Pascal.

Você pode estar pensando que as operações WAIT e SIGNAL se parecem com as SLEEP e WAKEUP, que vimos anteriormente como conduzindo a condições de corrida fatais. De fato, elas são muito semelhantes, mas com uma diferença crucial: SLEEP e WAKEUP falhavam porque, enquanto um processo estava tentando ir dormir, o outro estava tentando acordá-lo. Com monitores, isso não pode acontecer. A exclusão mútua automática em procedimentos de monitor garante que se, digamos, o produtor dentro de um procedimento de monitor descobre que o buffer está cheio, ele será capaz de completar a operação WAIT sem precisar preocupar-se com a possibilidade de o agendador poder alternar para o consumidor antes do WAIT completar-se. O consumidor não está nem mesmo será autorizado a entrar no monitor até WAIT terminar, e o produtor ser marcado como não mais executável.

Tornando automática a exclusão mútua de regiões críticas, os monitores tornaram a programação paralela muito menos sujeita a erros do que com semáforos. Mas eles ain-

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count – 1;
    if count = N – 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item
  end
end
```

Figura 2-14 Um esboço do problema produtores e consumidores com monitores. Só um procedimento de monitor está ativo por vez. O buffer tem *N* entradas.

da têm algumas desvantagens. Não é à-toa que a Figura 2-14 é escrita em um estranho Pascal em vez de em C, como são os outros exemplos neste livro. Como dissemos anteriormente, os monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e arranjar a exclusão mútua de algum modo. C, Pascal e a maioria das outras linguagens não têm monitores, então, não é razoável esperar que seus compiladores implementem qualquer regra de exclusão mútua. De fato, como o compilador poderia saber quais procedimentos estão em monitores e quais não estão?

Essas mesmas linguagens também não têm semáforos, mas adicionar semáforos é fácil: tudo que você precisa fazer é adicionar à biblioteca duas curtas rotinas em código *assembly* para produzir as chamadas de sistema UP e DOWN. Os compiladores sequer precisam saber que elas existem. Naturalmente, os sistemas operacionais precisam saber dos semáforos, mas, se você tem pelo menos um sistema operacional baseado em semáforo, você ainda pode escrever os programas de usuário para ele em C ou C + + (ou mesmo BASIC, se for masoquista o suficiente). Com monitores, você precisa de uma linguagem que os tenha incorporados. Algumas linguagens os têm, como a Concurrent Euclid (Holt, 1983), mas são raras.

Outro problema com monitores e também com semáforos é que eles foram projetados para resolver o problema de exclusão mútua em uma ou em mais CPUs que têm acesso a uma memória comum. Colocando os semáforos na memória compartilhada e protegendo-os com instruções TSL, podemos evitar as condições de corrida. Quando vamos para um sistema que consiste em múltiplas CPUs distribuídas, cada uma com sua própria memória privada, conectadas por uma rede local, essas primitivas tornam-se inaplicáveis. A conclusão é que semáforos são de muito baixo nível, e os monitores não são utilizáveis, exceto em algumas linguagens de programação. Além disso, nenhu-

ma das primitivas oferece troca de informações entre máquinas. Algo mais é necessário.

2.2.7 Passagem de Mensagem

Esse algo mais é a **passagem de mensagem.** Esse método de comunicação interprocesso utiliza duas primitivas SEND e RECEIVE que, como os semáforos e, ao contrário dos monitores, são chamadas de sistema em vez de construções da linguagem. Como tal, eles podem ser facilmente colocados em procedimentos de biblioteca, como

```
send(destination, &message);
```

e

```
receive(source, &message);
```

A primeira chamada envia uma mensagem (*message*) para um dado destino (*destination*) enquanto a segunda recebe uma mensagem de uma determinada origem (ou de qualquer, se o destinatário não se importar). Se nenhuma mensagem estiver disponível, ele poderia bloquear até uma chegar. Alternativamente, ele poderia retornar imediatamente com um código de erro.

Questões de Projeto para Sistemas de Passagem de Mensagem

Os sistemas de passagem de mensagem têm muitos problemas desafiadores e questões de projeto que não surgem com os semáforos nem com os monitores, especialmente se os processos comunicantes estiverem em máquinas diferentes conectadas por uma rede. Por exemplo, as mensagens podem perder-se na rede. Para evitar perda de mensagens, o remetente e o destinatário podem concordar que logo que a mensagem for recebida, o destinatário enviará de volta uma mensagem especial de **reconhecimento**. Se o remetente não receber o reconhecimento dentro de um certo intervalo de tempo, ele retransmite a mensagem.

Agora considere o que acontece se a mensagem em si é recebida corretamente, mas o reconhecimento perde-se. O remetente retransmitirá a mensagem, de modo que o destinatário irá recebê-la duas vezes. É essencial que o destinatário possa diferenciar entre uma nova mensagem e a retransmissão de uma antiga. Normalmente, esse problema é resolvido colocando-se uma seqüência de números consecutivos em cada mensagem original. Se o destinatário receber uma mensagem contendo o mesmo número de seqüência que a mensagem anterior, ele sabe que a mensagem é uma duplicata e que pode ser ignorada.

Os sistemas de mensagem também têm de lidar com a pergunta de como os processos são nomeados, para que o processo especificado em uma chamada SEND ou RECEIVE não seja ambíguo. A **autenticação** também é uma questão em sistemas de mensagem: como o cliente pode descobrir que está comunicando-se com o verdadeiro servidor de arquivos e não com um impostor?

Na outra extremidade do espectro, também há questões de projeto que são importantes quando o remetente e o destinatário estão na mesma máquina. Uma delas é o desempenho. Copiar mensagens de um processo para outro é sempre mais lento que fazer uma operação de semáforo ou entrar em um monitor. Muito trabalho foi aplicado para fazer a passagem de mensagens eficiente. Cheriton (1984), por exemplo, sugeriu limitar o tamanho da mensagem para um tamanho que caberá nos registradores da máquina, e, então, fazer a passagem da mensagem utilizando os registradores.

O Problema dos Produtores e dos Consumidores com a Passagem de Mensagem

Agora vejamos como o problema dos produtores e dos consumidores pode ser resolvido com passagem de mensagem e de memória não-compartilhada. Uma solução é dada na Figura 2-15. Supomos que todas as mensagens são do mesmo tamanho e que as mensagens enviadas, mas ainda não-recebidas, são armazenadas automaticamente pelo sistema operacional. Nessa solução, um total de N mensagens é utilizado, análogo às N entradas em um buffer de memória compartilhada. O consumidor inicia enviando N mensagens vazias para o produtor. Sempre que o produtor tem um item para dar ao consumidor, ele pega uma mensagem vazia e envia de volta uma cheia. Dessa maneira, o número total de mensagens no sistema permanece constante com o tempo, o que permite que elas sejam armazenadas em uma determinada quantidade de memória conhecida antes.

Se o produtor trabalha mais rapidamente que o consumidor, todas as mensagens acabarão cheias, esperando o consumidor; o produtor será bloqueado, esperando uma vazia voltar. Se o consumidor trabalhar mais rapidamente, então o inverso acontece: todas as mensagens estarão vazias esperando o produtor enchê-las; o consumidor será bloqueado, esperando uma mensagem cheia.

Muitas variantes são possíveis com passagem de mensagem. Para os iniciantes, vejamos como as mensagens são endereçadas. Uma maneira é atribuir a cada processo um endereço único e ter as mensagens endereçadas para processos. Uma maneira diferente é inventar uma nova estrutura de dados, chamada **caixa de correio** (*mailbox*). Uma caixa de correio é um lugar para armazenar um certo número de mensagens, em geral, especificado quando a caixa de correio é criada. Quando as caixas de correio são utilizadas, os parâmetros de endereço nas chamadas SEND e RECEIVE são caixas de correio, não processos. Quando um processo tenta enviar para uma caixa de correio que está cheia, ele é suspenso até que uma mensagem seja removida dessa caixa de correio, dando espaço para uma nova.

Para o problema dos produtores e dos consumidores, ambos, produtor e consumidor, criariam caixas de correio suficientemente grandes para armazenar N mensagens. O

```
#define N 100                              /* número de entradas no buffer */

void producer(void)
{
 int item;
 message m;                                /* buffer de mensagem */

while (TRUE) {
       produce_item(&item);                /* gera algo para colocar no buffer */
       receive(consumer, &m);              /* espera chegar uma vazia */
       build_message(&m, item);            /* constrói uma mensagem para enviar */
       send(consumer, &m);                 /* envia o item para o consumidor */
 }
}

void consumer(void)
{
 int item, i;
 message m;

 for (i = 0; i < N; i++) send(producer, &m);  /* envia N vazias */
 while (TRUE) {
       receive(producer, &m);              /* recebe a mensagem contendo o item */
       extract_item(&m, &item);            /* extrai o item da mensagem */
       send(producer, &m);                 /* envia de volta resposta vazia */
       consume_item(item);                 /* faz algo com o item */
 }
}
```

Figura 2-15 O problema dos produtores e dos consumidores com *N* mensagens.

produtor enviaria mensagens contendo dados para a caixa de correio do consumidor, e o consumidor enviaria mensagens vazias para a caixa de correio do produtor. Quando as caixas de correio são utilizadas, o mecanismo de armazenamento é claro: a caixa de correio do destino armazena mensagens que foram enviadas para o processo de destino mas ainda não foram aceitas.

O outro extremo de ter caixas de correio é eliminar todo armazenamento. Quando essa abordagem é adotada, se o SEND é feito antes do RECEIVE, o processo de envio é bloqueado até que o RECEIVE aconteça, momento em que a mensagem pode ser copiada diretamente do remetente para o destinatário, sem nenhum armazenamento intermediário. De maneira semelhante, se o RECEIVE é feito primeiro, o destinatário é bloqueado até que um SEND aconteça. Essa estratégia é freqüentemente conhecida como **rendez-vous.** É mais fácil de implementar do que um esquema de mensagens armazenadas mas é menos flexível porque o remetente e o destinatário são forçados a executar em *lockstep*, i. e., intercalando ação e bloqueio, de acordo com o ritmo do processo mais lento.

A comunicação entre processos de usuário no MINIX (e no UNIX) é via *pipes*, que são, efetivamente, caixas de correio. A única diferença real entre um sistema de mensagem com caixas de correio e o mecanismo de *pipes* é que os *pipes* não conservam os limites da mensagem. Em outras palavras, se um processo gravar 10 mensagens de 100 bytes em um *pipe* e outro processo ler 1.000 bytes desse *pipe*, o leitor receberá todas as 10 mensagens imediatamente. Com um verdadeiro sistema de mensagens, cada READ deve retornar só uma mensagem. Naturalmente, se os processos concordarem em sempre ler e gravar mensagens de tamanho fixo no *pipe*, ou em finalizar cada mensagem com um caractere especial (p. ex., uma quebra de linha), não haverá nenhum problema. Os processos que constituem o sistema operacional MINIX utilizam um verdadeiro esquema de mensagens de tamanho fixo para comunicação entre eles.

2.3 PROBLEMAS CLÁSSICOS DE CIP

A literatura de sistemas operacionais está repleta de problemas interessantes que foram amplamente discutidos e analisados. Nas seções a seguir, examinaremos três dos problemas mais conhecidos.

2.3.1 O Problema dos Filósofos Jantando

Em 1965, Dijkstra propôs e resolveu um problema de sincronização que ele chamou de **problema dos filósofos jantando.** Desde essa época, todo mundo que inventava outra primitiva de sincronização sentia-se obrigado a demonstrar o quão maravilhosa a nova primitiva era, mostrando o quão elegantemente ela resolvia o problema dos filósofos jantando. O problema pode ser exposto de uma maneira simples, como segue. Cinco filósofos estão senta-

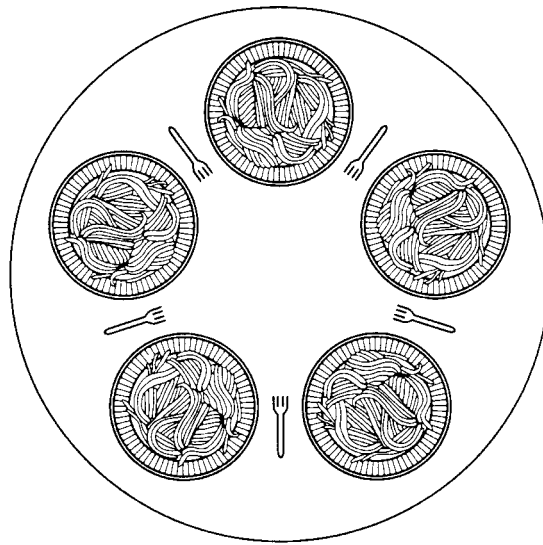



Figura 2-16 Hora do jantar no Departamento de Filosofia.

dos ao redor de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete está tão escorregadio que o filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos está um garfo. A organização da mesa está ilustrada na Figura 2-16.

A vida de um filósofo consiste em períodos alternados de comer e de pensar. (Isso é uma abstração, mesmo para filósofos, mas as demais atividades são irrelevantes aqui.) Quando um filósofo fica com fome, ele tenta pegar os garfos da esquerda e da direita, um de cada vez, em qualquer ordem. Se tiver sucesso em pegar dois garfos, ele come por um tempo e, então, coloca os garfos na mesa e continua a pensar. A pergunta-chave é: você pode escrever um programa para cada filósofo fazer o que se supõe que ele deve fazer e nunca ficar imobilizado? (Foi indicado que o requisito de dois garfos é um pouco artificial; talvez devêssemos trocar a comida italiana por comida chinesa, substituindo o espaguete por arroz, e os garfos por pauzinhos.)

A Figura 2-17 mostra a solução óbvia. O procedimento *take_fork* espera até que o garfo especificado esteja disponível e, então, pega-o. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente. Nenhum será capaz de pegar seus garfos direitos e haverá um impasse.

Poderíamos modificar o programa de modo que depois de pegar o garfo esquerdo, o programa verificasse se o garfo direito está disponível. Se não estiver, o filósofo coloca na mesa o esquerdo, espera algum tempo e, então, repete o processo inteiro. Essa proposta também fracassa, embora por uma razão diferente. Com um pouquinho de má sorte, todos os filósofos poderiam iniciar o algoritmo simultaneamente, pegando seus garfos esquerdos, vendo que seus garfos direitos não estão disponíveis, colocando na mesa seus garfos esquerdos, esperando, pegando seus garfos esquerdos novamente simultaneamente e assim por diante, eternamente. Uma situação como essa, em que todos os programas continuam a executar indefinidamente mas não conseguem fazer qualquer progresso é chamada **fome.** (E chama-se fome mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Agora você pode pensar: "Se os filósofos apenas esperassem um tempo aleatório em vez do mesmo tempo após não conseguir pegar o garfo da mão direita, a chance de que tudo continuaria em *lockstep* por um longo período é muito pequena". Essa observação é verdadeira, mas em alguns aplicativos talvez se prefira uma solução que sempre funciona e não possa falhar devido a uma série improvável de números aleatórios. (Pense no controle de segurança em uma usina nuclear.)

Um aprimoramento na Figura 2-17 que não resulta em impasse nem em fome é proteger as cinco declarações que se seguem à chamada a *think* com um semáforo binário. Antes de começar a pegar os garfos, um filósofo faria um DOWN em *mutex*. Depois de substituir os garfos, ele faria um UP em *mutex*. De um ponto de vista teórico, essa solução é adequada. De um ponto de vista prático, ela apresenta uma falha em termos de desempenho: só um filósofo pode estar comendo de cada vez. Com cinco garfos disponíveis, deveríamos ser capazes de permitir que dois filósofos comessem ao mesmo tempo.

A solução apresentada na Figura 2-18 é correta e também permite o paralelismo máximo para um número arbitrário de filósofos. Ela utiliza uma matriz, *state*, para monitorar se um filósofo está comendo, se está pensando, ou se está com fome (tentando pegar garfos). Um filósofo só pode passar para o estado "comendo" se nenhum vizi-

```
#define N 5                       /* número de filósofos *!

void philosopher(int i)           /* número do filósofo, de 0 a 4 */
{
 while (TRUE) {
        think();                  /* filósofo está pensando */
        take_fork(i);             /* pega o garfo esquerdo */
        take_fork((i+1 ) % N);    /* pega o garfo direito; % é o operador de módulo */
        eat();                    /* nham-nham, espaguete */
        put fork(i);              /* coloca o garfo esquerdo de volta na mesa */
        put fork((i+1 ) % N);     */ coloca o garfo direito de volta na mesa */
 }
}
```

Figura 2-17 Uma "não-solução" para o problema dos filósofos jantando.

nho estiver comendo. Os vizinhos do filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se o *i* é 2, *LEFT* é 1 e *RIGHT* é 3.

O programa utiliza uma matriz de semáforos, um por filósofo, assim os filósofos com fome podem bloquear se os garfos necessários estiverem ocupados. Note que cada processo executa o procedimento *philosopher* como seu código principal. Mas, *take_forks*, *put_forks* e *test*, são procedimentos comuns e não processos separados.

2.3.2 O Problema dos Leitores e dos Escritores

O problema de filósofos jantando é útil para modelar processos que estão competindo por acesso exclusivo a um número limitado de recursos, como dispositivos de E/S. Outro problema famoso é o problema dos leitores e dos escritores (Courtois *et al.*, 1971), que modela o acesso a um banco de dados. Imagine, por exemplo, sistema de reservas de uma companhia aérea, com muitos processos concorrentes querendo ler e escrever. É permitido ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas, se um processo estiver atualizando (escrevendo) o banco de dados, nenhum outro processo poderá ter acesso ao banco de dados, nem mesmo leitores. A pergunta é: como você programa os leitores e os escritores? Uma solução é mostrada na Figura 2-19.

Nesta solução, o primeiro leitor que recebe acesso ao banco de dados faz um DOWN no semáforo *db*. Os leitores subseqüentes meramente incrementam um contador *rc*. À medida que os leitores saem, eles decrementam o contador e o último faz um UP no semáforo, permitindo que um escritor bloqueado, se houver um, entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale comentar. Suponha que enquanto um leitor esteja usando o banco de dados, outro leitor apareça. Como ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Um terceiro leitor e os leitores subseqüentes também podem ser admitidos, se aparecerem.

Agora suponha que um escritor apareça. O escritor não pode ser admitido no banco de dados, uma vez que os escritores devem ter acesso exclusivo. Então, o escritor é suspenso. Mais tarde, aparecem outros leitores. Contanto que pelo menos um leitor ainda esteja ativo, os leitores subseqüentes são admitidos. Como uma conseqüência dessa estratégia, contanto que haja um estoque constante de leitores, todos eles entrarão logo que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor chegar, digamos, a cada 2 segundos, e cada leitor levar 5 segundos para fazer seu trabalho, o escritor nunca entrará.

Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar os leitores que estavam ativos quando ele chegou para terminar, mas não precisa esperar leitores que vieram depois dele. A desvantagem dessa solução é que ela obtém menor paralelismo e, portanto, tem desempenho inferior. Courtois e outros apresentaram uma solução que dá prioridade aos escritores. Para detalhes, sugerimos a leitura do seu paper.

2.3.3 O Problema do Barbeiro Adormecido

Outro problema clássico de CIP acontece em uma barbearia. A barbearia tem um barbeiro, uma cadeira de barbeiro e *n* cadeiras para os clientes esperarem, se houver algum, sentados. Se não houver nenhum cliente presente, o barbeiro senta-se na sua cadeira e dorme, como ilustrado na Figura 2-20. Quando um cliente chega, ele tem de acordar o barbeiro. Se outros clientes chegarem enquanto o barbeiro estiver cortando o cabelo de um cliente, eles sentam-se (se houver cadeiras vazias) ou saem da barbearia (se todas as cadeiras estiverem cheias). O problema é programar o barbeiro e os clientes sem cair em condições de corrida.

Nossa solução utiliza três semáforos: *customers*, que conta os clientes que esperam (excluindo o cliente na cadeira do barbeiro, que não está esperando), *barbers*, o número de barbeiros que estão desocupados, esperando clientes (0 ou 1) e *mutex*, que é utilizado para exclusão mú-

```
#define N           5         /* número de filósofos */
#define LEFT        (i-1)%N   /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N   /* número do vizinho à direita de i */
#define THINKING    0         /* filósofo está pensando */
#define HUNGRY      1         /* filósofo está tentando pegar os garfos */
#define EATING      2         /* filósofo está comendo */

typedef int semaphore;        /* semáforos são um tipo especial de int */
int state[N];                 /* matriz para monitorar o estado de todos */
semaphore mutex = 1;          /* exclusão mútua para regiões críticas */
semaphore s[N];               /* um semáforo por filósofo */

void philosopher(int i)       /* i: número do filósofo, de 0 a N-1 */
{
 while (TRUE)                 /* repete eternamente */
       { think();             /* filósofo está pensando */
       take_forks(i);         /* pega dois garfos ou bloqueia */
       eat();                 /* nham-nham, espaguete */
       put_forks(i);          /* coloca de volta os garfos na mesa */
 }
}

void take_forks(int i)        /* i: número do filósofo, de 0 a N-1 */
{
 down(&mutex);                /* entra na região crítica */
 state[i] = HUNGRY;           /* registra o fato de que o filósofo está com fome */
 test(i);                     /* tenta pegar 2 garfos */
 up(&mutex);                  /* sai da região crítica */
 down(&s[i]);                 /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)             /* i: número do filósofo, de 0 a N-1 */
{
 down(&mutex);                /* entra na região crítica */
 state[i] = THINKING;         /* filósofo terminou de comer */
 test(LEFT);                  /* vê se o vizinho esquerdo agora pode comer */
 test(RIGHT);                 /* vê se o vizinho direito agora pode comer */
 up(&mutex);                  /* sai da região crítica */
}

void test(i)                  /* i: número do filósofo, de 0 a N-1 */
{
 if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
       state[i] = EATING;
       up(&s[i]);
 }
}
```

Figura 2-18 Uma solução para o problema dos filósofos jantando.

tua. Também precisamos de uma variável, *waiting*, que também conta os clientes que esperam. É essencialmente uma cópia de *customers*. A razão para ter *waiting* é que não há nenhum meio de ler o valor atual de um semáforo. Nessa solução, um cliente que entra na loja tem de verificar o número de clientes que estão esperando. Se for inferior ao número de cadeiras, ele permanece: caso contrário, sai.

Nossa solução é mostrada na Figura 2-21. Quando o barbeiro chega para trabalhar de manhã, ele executa o procedimento *barber*, o que o faz bloquear no semáforo *customers* até alguém chegar. Ele, então, vai dormir como mostrado na Figura 2-20.

Quando um cliente chega, ele executa *customer*, começando por adquirir *mutex* para entrar em uma região crítica. Se outro cliente entrar logo depois, não será capaz de fazer qualquer coisa até que o primeiro tenha liberado *mutex*. O cliente, então, verifica se o número de clientes esperando é inferior ao número de cadeiras. Se não, libera *mutex* e sai sem cortar o cabelo.

Se houver uma cadeira disponível, o cliente incrementa a variável de número inteiro *waiting*. Então, ele faz um UP no semáforo *customers*, acordando assim o barbeiro. Nesse ponto, o cliente e o barbeiro estão ambos acordados. Quando o cliente libera *mutex*, o barbeiro o pega, faz alguma arrumação e começa o corte de cabelo.

```
typedef int semaphore;              /* utilize sua imaginação */
semaphore mutex = 1;                /* controla o acesso a 'rc' */
semaphore db = 1;                   /* controla o acesso ao banco de dados */
intrc = 0;                          /* número de processos lendo ou querendo ler */

void reader(void)
{
 while (TRUE) {                     /* repete eternamente */
        down(&mutex);               /* obtém acesso exclusivo a 'rc' */
        rc = rc + 1;                /* mais um leitor agora */
        if (rc == 1) down(&db);     /* se este for o primeiro leitor ... */
        up(&mutex);                 /* libera acesso exclusivo a 'rc' */
        read_data base();           /* acessa os dados */
        down(&mutex);               /* obtém acesso exclusivo a 'rc' */
        rc = rc - 1;                /* um leitor a menos agora */
        if (rc == 0) up(&db);       /* se este for o último leitor ... */
        up(&mutex);                 /* libera acesso exclusivo a 'rc' */
        use_data_read();            /* região não-crítica */
 }
}

writer(void)
{
 while (TRUE) {                     /* repete eternamente */
        { think_up_data();          /* região não-crítica */
        down(&db);                  /* obtém acesso exclusivo */
        write_data_base();          /* atualiza os dados */
        up(&db);                    /* libera acesso exclusivo */
 }
}
```

Figura 2-19 Uma solução para o problema dos leitores e dos escritores.

Figura 2-20 O barbeiro adormecido.

Quando o corte de cabelo termina, o cliente sai do procedimento e sai da barbearia. Diferentemente de nossos exemplos anteriores, não há nenhum laço para o cliente porque cada um recebe só um corte de cabelo. O barbeiro, entretanto, faz um laço, tentando receber o próximo cliente. Se um cliente estiver presente, outro corte de cabelo é feito. Se não, o barbeiro vai dormir

A propósito, vale indicar que embora os problemas dos leitores e dos escritores e o do barbeiro adormecido não envolvam transferência de dados, eles ainda pertencem à área de CIP porque envolvem sincronização entre múltiplos processos.

2.4 AGENDAMENTO DE PROCESSO

Nos exemplos das seções anteriores, com freqüência tivemos situações em que dois ou mais processos (p. ex., produtor e consumidor) eram logicamente executáveis. Quando mais de um processo é executável, o sistema operacional deve decidir qual executar primeiro. A parte do sistema operacional que toma essa decisão é chamada **agendador;** o algoritmo que ele utiliza é chamado **algoritmo de agendamento.**

De volta aos velhos dias dos sistemas de lote com entrada na forma de imagens de cartões em uma fita magnética, o algoritmo de agendamento era simples: apenas execute o próximo trabalho na fila. Com sistemas de compartilhamento de tempo, o algoritmo de agendamento é mais complexo, uma vez que, freqüentemente, há múltiplos usuários esperando serviço e pode haver um ou mais fluxos de lote também (p. ex. em uma companhia de seguro, para processar pedidos). Mesmo em computadores pessoais, pode haver vários processos iniciados pelo usuário competindo pela CPU, para não mencionar trabalhos em segundo plano, como servidores de rede ou correio eletrônico enviando ou recebendo mensagens.

Antes de examinar algoritmos de agendamento específicos, devemos pensar no que o agendador está tentando obter. Afinal de contas, o agendador está preocupado em decidir sobre política, não em fornecer um mecanismo. Vários critérios vêm à mente sobre o que constitui um bom algoritmo de agendamento. Algumas possibilidades incluem:

```
#define CHAIRS 5                 /* número de cadeiras para esperar clientes */

typedef int semaphore;           /* utilize sua imaginação */

semaphore customers = 0;         /* número de clientes que esperam serviço */
semaphore barbers = 0;           /* número de barbeiros que esperam clientes */
semaphore mutex = 1;             /* para exclusão mútua */
int waiting = 0;                 /* clientes estão esperando (não cortando cabelo) */

void barber(void)
{
 while (TRUE) {
        down(customers);         /* vai dormir se o número de clientes for 0 */
        down(mutex);             /* adquire acesso a 'waiting' */
        waiting = waiting - 1;   /* decrementa a conta de clientes esperando */
        up(barbers);             /* um barbeiro está agora pronto para cortar o cabelo */
        up(mutex);               /* libera 'waiting' */
        cut hair();              /* corta o cabelo (fora da região crítica) */
 }
}

void customer(void)
{
 down(mutex);                    /* entra na região crítica */
 if (waiting < CHAIRS) {         /* se não houver cadeiras livres, vai embora */
        waiting = waiting + 1;   /* incrementa a contagem de clientes esperando */
        up(customers);           /* acorda o barbeiro se necessário */
        up(mutex);               /* libera o acesso a 'waiting' */
        down (barbers) ;         /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut();           /* senta-se e é atendido */
 } else {
        up(mutex);               /* a barbearia está cheia; não espera */
 }
}
```

Figura 2-21 Uma solução para o problema do barbeiro adormecido.

1. A imparcialidade — assegura que cada processo receba sua justa parte da CPU.
2. A eficiência — mantém a CPU ocupada 100% do tempo.
3. Tempo de resposta — minimiza o tempo de resposta para usuários interativos.
4. *Turnaround* — minimiza o tempo que os usuários de lote devem esperar pela saída.
5. *Throughput* — maximiza o número de *jobs* processados por hora.

Uma rápida análise mostrará que algumas dessas metas são contraditórias. Para minimizar o tempo de resposta para usuários interativos, o agendador não deve executar nenhum trabalho de lote (exceto talvez entre 3 e 6h, quando todos os usuários interativos estão confortáveis em suas camas). Mas os usuários de lote provavelmente não gostarão desse algoritmo; ele viola o critério 4. Pode ser demonstrado (Kleinrock, 1975) que qualquer algoritmo de agendamento que favorece alguma classe de trabalho acaba prejudicando outra classe de trabalho. A quantidade de tempo de CPU disponível é finita, afinal de contas. Para dar mais a um usuário, você tem de dar menos a outro usuário. Assim é a vida.

Uma complicação com que os agendadores têm de lidar é que cada processo é único e imprevisível. Alguns gastam muito tempo esperando E/S de arquivo, enquanto outros utilizariam a CPU durante horas por vez se lhe fosse dada a chance. Quando o agendador começa a executar algum processo, ele nunca sabe com certeza quanto tempo levará até que o processo bloqueie, seja para E/S, um semáforo ou por qualquer outra razão. Para certificar-se de que nenhum processo executará por muito tempo, quase todos os computadores têm um temporizador ou um relógio eletrônico interno, que causa uma interrupção periodicamente. Uma freqüência de 50 ou 60 vezes por segundo (chamada 50 ou 60 Hertz e abreviado como Hz) é comum, mas em muitos computadores o sistema operacional pode configurar a freqüência de temporização para qualquer valor que ele quiser. Em cada interrupção de relógio, o sistema operacional começa a executar e decide se o processo atualmente em execução deve ter permissão para continuar, ou se teve tempo suficiente de CPU até o momento e deve ser suspenso para dar a CPU a outro processo.

A estratégia de permitir processos que são logicamente executáveis serem temporariamente suspensos é chamada **agendamento preemptivo**, e está em oposição ao método de **executar até concluir** dos antigos sistemas de lote. Esse método também é chamado **agendamento não-preemptivo**. Como vimos ao longo deste capítulo, um processo pode ser suspenso em um instante arbitrário, sem aviso, para que outro processo possa ser executado. Isso leva a condições de corrida e precisa de semáforos, de monitores, de mensagens ou de algum outro método sofisticado para evitá-las. Por outro lado, a política de deixar um processo executar o quanto quiser significaria que um processo calculando π até um bilhão de casas poderia negar serviço a todos os outros processos indefinidamente.

Portanto, embora algoritmos de agendamento não-preemptivo sejam simples e fáceis de implementar, eles normalmente não são adequados para sistemas de propósito geral com múltiplos usuários competindo. Por outro lado, para um sistema dedicado, como um servidor de banco de dados, pode ser bastante razoável para o processo-mestre iniciar um processo-filho trabalhando em uma requisição e deixá-lo executar até que o processo se complete ou bloqueie. A diferença do sistema de propósito geral é que todos os processos no sistema de banco de dados estão sob o controle de um único mestre, que sabe o que cada filho fará e quanto tempo levará.

2.4.1 Agendamento *Round Robin*

Agora vejamos alguns algoritmos de agendamento específicos. Um dos algoritmos mais simples, mais antigos e mais amplamente utilizados é o ***round robin***. A cada processo é atribuído um intervalo de tempo, chamado de seu **quantum**, durante o qual lhe é permitido executar. Se o processo ainda está executando no fim do quantum, é feita a preempção da CPU e ela é dada a outro processo. Se o processo bloqueou ou terminou antes de o quantum ter passado, é feita a comutação da CPU quando o processo bloqueia, naturalmente. O *round robin* é fácil de implementar. Tudo que precisamos que o agendador faça é manter uma lista de processos executáveis, como mostrado na Figura 2-22(a). Quando o processo utiliza todo seu quantum, ele é posto no fim da lista, como mostrado na Figura 2-22(b)

A única questão interessante relacionada com o *round robin* é a duração do quantum. Alternar de um processo para outro requer uma certa quantidade de tempo para fazer

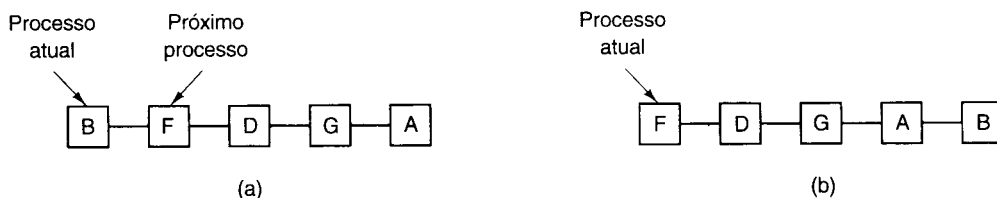



Figura 2-22 Agendamento *round robin*. (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que *B* utiliza todo seu quantum.

a administração — salvar e carregar registradores e mapas de memória, atualizar várias tabelas e listas, etc. Suponha que essa **comutação de processo** ou **comutação de contexto,** como às vezes é chamada, leve 5ms. Também suponha que o quantum é definido como 20ms. Com esses parâmetros, depois de fazer 20ms de trabalho útil, a CPU gastará 5ms na comutação de processos. Vinte por cento do tempo da CPU serão desperdiçados em trabalho administrativo.

Para melhorar a eficiência da CPU, poderíamos configurar o quantum como, digamos, 500ms. Agora, o tempo desperdiçado é inferior a 1%. Mas considere o que acontece em um sistema de compartilhamento de tempo se dez usuários interativos pressionarem a tecla Enter mais ou menos ao mesmo tempo. Dez processos serão postos na lista de processos executáveis. Se a CPU estiver desocupada, o primeiro iniciará imediatamente, o segundo não poderá iniciará até 1/2 s mais tarde e assim por diante. O infeliz último da fila irá esperar 5 s antes de ter uma chance, supondo que todos os outros utilizam seus quanta inteiros. A maioria dos usuários achará horrível uma resposta de 5 s para um comando. O mesmo problema pode ocorrer em um computador pessoal que suporta multiprogramação.

A conclusão pode ser formulada como segue: configurando o quantum com um tempo muito pequeno, ocorrem muitas comutações de processo, e a eficiência da CPU é reduzida; mas configurando-o com um tempo muito longo, a resposta pode ser pobre para curtas requisições interativas. Um quantum em torno de 100ms é freqüentemente um compromisso razoável.

2.4.2 Agendamento por Prioridade

O agendamento *round robin* faz a suposição implícita de que todo processo é igualmente importante. Com freqüência, as pessoas que possuem e operam computadores multiusuários têm idéias diferentes sobre tal assunto. Em uma universidade, a ordem de prioridade pode ser diretores primeiro, depois professores, secretários, inspetores e, por fim, os alunos. A necessidade de levar em conta fatores externos conduz ao **agendamento por prioridade.** A idéia básica é simples e direta: a cada processo é atribuído uma prioridade, e o processo executável com a maior prioridade recebe permissão para executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns mais importantes que outros. Por exemplo, a um processo de servidor que envia uma mensagem de correio eletrônico em segundo plano deve ser atribuído uma prioridade mais baixa que a um processo que exibe um vídeo na tela em tempo real.

Para evitar que processos de alta prioridade executem indefinidamente, o agendador pode diminuir a prioridade do processo atualmente em execução em cada tique de relógio (i. e., a cada interrupção de relógio). Se essa ação fizer com que sua prioridade caia para baixo do próximo processo com maior prioridade, ocorre uma comutação de processo. Alternativamente, a cada processo pode ser atribuído um quantum máximo em que lhe é permitido usar a CPU continuamente. Quando esse quantum é utilizado, é dada uma chance de executar ao próximo processo com maior prioridade.

As prioridades podem ser atribuídas aos processos estática ou dinamicamente. Em um computador militar, o processo iniciado pelo general pode começar com prioridade 100, processos iniciados por coronéis com 90, majores com 80, capitães com 70, tenentes com 60 e assim por diante. Alternativamente, em um centro de computação comercial, trabalhos de alta prioridade podem custar 100 dólares por hora; os de média prioridade, 75 dólares por hora; e os de baixa prioridade 50 dólares por hora. O sistema UNIX tem um comando, *nice*, que permite que um usuário voluntariamente reduza a prioridade do seu processo, para ser gentil (*nice*) com os outros usuários. Ninguém o utiliza.

As prioridades também podem ser atribuídas dinamicamente pelo sistema para atingir certas metas do sistema. Por exemplo, alguns processos são altamente associados a E/S e gastam a maior parte do seu tempo esperando a E/S completar-se. Sempre que esse processo quiser a CPU, ele deve receber a CPU imediatamente, para deixá-lo iniciar sua próxima requisição de E/S, que pode, então, proceder em paralelo com outro processo que realmente faz computação. Deixando o processo associado a E/S esperar por muito tempo pela CPU somente significará fazê-lo ocupar memória por um tempo desnecessariamente longo. Um algoritmo simples para oferecer bom serviço a processos associados a E/S é configurar a prioridade como $1/f$, onde f é a fração do último quantum que o processo utilizou. Um processo que utilizou só 2ms de seus 100ms de quantum receberia prioridade 50, enquanto um processo que executasse 50ms antes de bloquear receberia prioridade 2 e um processo que utilizou o quantum inteiro receberia prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e utilizar agendamento por prioridade entre as classes, mas agendamento por *round robin* dentro de cada classe. A Figura 2-23 mostra um sistema com quatro classes de prioridade. O algoritmo de agendamento é como segue: contanto que haja processos executáveis em classe de prioridade 4, apenas execute cada um pelo tempo do seu quantum, no modo *round robin* e nunca se incomode com classes de prioridade mais baixas. Se a classe de prioridade 4 estiver vazia, então, execute os processos classe 3 por *round robin*. Se as classes 4 e 3 estiverem vazias, então, execute a classe 2 por *round robin* e assim por diante. Se as prioridades não forem ajustadas ocasionalmente, as classes de prioridade mais baixa podem passar fome até morrer.

2.4.3 Múltiplas Filas

Um dos agendadores de prioridade mais antigos estava no CTSS (Corbato *et al.*, 1962). O CTSS tinha o problema de que a comutação de processos era muito lenta porque

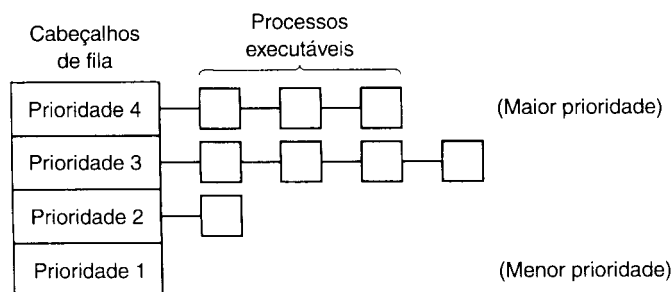



Figura 2-23 Um algoritmo de agendamento com quatro classes de prioridade.

os 7094 podiam armazenar só um processo na memória. Cada comutação significava enviar o processo atual para o disco e ler um novo processo do disco. Os projetistas do CTSS rapidamente reconheceram que era mais eficiente dar um quantum grande para processos associados à CPU de vez em quando, no lugar de dar pequenos quanta com freqüência (reduzindo as trocas). Por outro lado, dar a todo processo um quantum grande poderia levar a um péssimo tempo de resposta, como já vimos. Sua solução foi definir classes de prioridade. Processos de classe mais alta eram executados pelo tempo de um quantum. Processos na classe de prioridade mais alta seguinte eram executados por dois quanta. Os processos na próxima classe eram executados por quatro quanta e assim por diante. Sempre que um processo utilizava todos os quanta permitidos a ele, era movido para baixo uma classe.

Como um exemplo, considere um processo que precisasse computar continuamente por 100 quanta. Inicialmente lhe seria dado um quantum, então, fazia-se sua troca para o disco. Da próxima vez, ele receberia dois quanta antes de ser feita sua troca a partir do disco. Em sucessivas execuções, ele poderia receber 4, 8, 16, 32 e 64 quanta, embora tivesse utilizado apenas 37 dos 64 quanta totais para completar seu trabalho. Só 7 trocas seriam necessárias (incluindo a carga inicial) em vez de 100 com um algoritmo de *round robin* puro. Além disso, à medida que o processo afundava cada vez mais nas filas de prioridade, ele seria executado com cada vez menos freqüência, poupando a CPU para processos curtos interativos.

A seguinte política foi adotada para evitar que um processo que necessitava executar durante muito tempo ao ser iniciado pela primeira vez, mas tornava-se interativo mais tarde, fosse punido eternamente. Sempre que uma tecla Enter era pressionada em um terminal, o processo pertencente a esse terminal era movido para a classe de prioridade mais alta, na suposição de que ele estava para tornar-se interativo. Um belo dia, um usuário com um processo intensamente associado à CPU descobriu que o simples fato de pressionar Enter várias vezes, aleatoriamente, melhorava seu tempo de resposta. Então, ele contou isso a todos os seus amigos. Moral da história: o funcionamento na prática é muito mais difícil do que o funcionamento em princípio.

Muitos outros algoritmos foram utilizados para atribuir classes de prioridade a processos. Por exemplo, o influente sistema XDS 940 (Lampson, 1968), construído em Berkeley, tinha quatro classes de prioridade, chamadas terminal, E/S, quantum breve e quantum longo. Quando um processo que estava esperando uma entrada de terminal era finalmente acordado, ele entrava na classe de maior prioridade (terminal). Quando um processo esperando um bloco de disco tornava-se pronto, ele entrava na segunda classe. Quando um processo ainda estava executando quando seu quantum era ultrapassado, ele era inicialmente colocado na terceira classe. Entretanto, se um processo gastava seu quantum muitas vezes, enquanto estava na fila sem bloquear para E/S de terminal ou outra, ele era movido para baixo na fila. Muitos outros sistemas utilizam algo semelhante para favorecer usuários interativos sobre processos em segundo plano.

2.4.4 *Job* Mais Curto Primeiro

A maioria dos algoritmos anteriores foi projetada para sistemas interativos. Agora examinemos um que é especialmente apropriado a *jobs* em lote, para os quais o tempo de execução é conhecido de antemão. Em uma companhia de seguros, por exemplo, as pessoas podem prever exatamente quanto tempo levará para executar um lote de 1.000 pedidos, uma vez que um trabalho semelhante é feito todos os dias. Quando vários *jobs* igualmente importantes estão na fila de entrada esperando ser iniciados, o agendador deve utilizar ***job* mais curto primeiro.** Veja a Figura 2-24. Aqui encontramos quatro *jobs* A, B, C e D com tempos de execução de 8, 4, 4 e 4 minutos, respectivamente. Executando-os nessa ordem, o tempo de retorno para A é 8 minutos, para B é 12, para C é 16 e para D é 20 minutos para uma média de 14 minutos.

Agora consideremos a execução desses quatro *jobs* utilizando *job* mais curto primeiro, como mostrado na Figura 2-24(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos para uma média de 11 minutos. *Job* mais curto primeiro é provavelmente ótimo. Considere o caso de quatro *jobs*, com tempos de execução de a, b, c e d, respectivamente. O primeiro *job* acaba no tempo a, o segundo acaba

Figura 2-24 Um exemplo de agendamento de *job* mais curto primeiro.

no tempo b e assim por diante. O tempo de retorno médio é $(4a + 3b + 2c + d)/4$. É claro que um contribui mais para a média que os outros tempos, então, ele deve ser o trabalho mais curto, com b vindo em seguida, depois c e, por fim, d como o mais longo na medida que ele afeta apenas o seu próprio tempo de retorno. O mesmo argumento aplica-se igualmente bem a qualquer número de *jobs*.

Como o *job* mais curto primeiro sempre produz o menor tempo médio de resposta, seria ótimo se ele pudesse ser utilizado para processos interativos também. Em certa medida, ele pode ser utilizado. Processos interativos, em geral, seguem o padrão de esperar comando, executar comando, esperar comando, executar comando e assim por diante. Se considerarmos a execução de cada comando como um "trabalho" separado, então poderíamos minimizar o tempo total de resposta para executar o mais curto primeiro. O único problema é descobrir qual dos processos atualmente executáveis é o mais curto.

Uma abordagem é fazer estimativas com base no comportamento passado e executar o processo com o menor tempo de execução estimado. Suponha que o tempo estimado por comando para um terminal seja T_0. Agora suponha que sua próxima execução seja medida como T_1. Poderíamos atualizar nossa estimativa pegando uma soma ponderada desses dois números, isto é, $aT_0 + (1-a)T_1$. Pela escolha de a podemos fazer o processo de estimativa ignorar execuções antigas rapidamente, ou lembrar delas durante muito tempo. Com $a = 1/2$, obtemos sucessivas estimativas de

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_0/8 + T_2/4 + T_3/2$$

Após três novas execuções, o peso de T_0 na nova estimativa caiu para 1/8.

A técnica de estimar o próximo valor em uma série pegando a média ponderada do valor atual medido e a estimativa anterior é, às vezes, chamada **envelhecimento**. Ela é aplicável a muitas situações nas quais uma previsão deve ser feita com base em valores anteriores. O envelhecimento é especialmente fácil de implementar quando $a = 1/2$. Tudo que é necessário é adicionar o novo valor à estimativa atual e dividir a soma por 2 (rotacionando-o 1 bit para a direita).

Vale indicar que o algoritmo do *job* mais curto primeiro só é ótimo quando todos os *jobs* estão disponíveis simultaneamente. Como um contra exemplo, considere cinco *jobs*, de A até E, com tempos de execução de 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3.

Inicialmente, só A ou B pode ser escolhido, uma vez que os outros três *jobs* não chegaram ainda. Utilizando *job* mais curto primeiro, executaremos os *jobs* na ordem A, B, C, D, E, para uma espera média de 4,6. Entretanto, executando-os na ordem B, C, D, E, A, obtém-se uma espera média de 4,4.

2.4.5 Agendamento Garantido

Uma abordagem completamente diferente para agendamento é fazer promessas realistas aos usuários sobre o desempenho e, então, conviver com elas. Uma promessa que é realista e fácil de cumprir é: se houver n usuários conectados no momento em que você estiver trabalhando, você receberá aproximadamente $1/n$ do poder da CPU. De maneira semelhante, em um sistema monousuário com n processos executando, todas as coisas sendo iguais, cada uma deve receber $1/n$ dos ciclos da CPU.

Para cumprir essa promessa, o sistema deve monitorar quanto da CPU cada processo teve, desde sua criação. Então, ele calcula quanto da CPU é atribuído a cada um, isto é, o tempo desde a criação dividido por n. Como a quantidade de tempo de CPU que cada processo realmente teve também é conhecida, é simples calcular a proporção entre o tempo real da CPU e o tempo da CPU atribuído. Uma proporção de 0,5 significa que um processo só teve metade do que devia ter tido e uma proporção de 2,0 significa que um processo teve o dobro do tempo que lhe foi atribuído. O algoritmo, então, executará o processo com a proporção mais baixa até que sua proporção tenha subido acima do seu competidor mais próximo.

2.4.6 Agendamento por Sorteio

Embora fazer promessas aos usuários e conviver com elas seja uma boa idéia, é difícil implementá-las. Mas outro algoritmo pode ser utilizado para dar resultados previsíveis de maneira semelhante com uma implementação muito mais simples. Ele é chamado **agendamento por sorteio** (Waldspurger e Weihl, 1994).

A idéia básica é dar bilhetes de loteria de processos aos vários recursos do sistema, como tempo de CPU. Sempre que uma decisão de agendamento tiver de ser feita, um bilhete de loteria é escolhido aleatoriamente, e o processo

que armazena esse bilhete recebe o recurso. Quando aplicado ao agendamento da CPU, o sistema pode realizar sorteios 50 vezes por segundo, com cada vencedor recebendo 20ms do tempo da CPU como prêmio.

Parafraseando George Orwell, "todos os processos são iguais, mas alguns processo são mais iguais". Os processos mais importantes podem receber bilhetes extras, para aumentar suas chances de ganhar. Se houver 100 bilhetes e um processo destacado receber 20 deles, ele terá 20% de chance de ganhar cada loteria. A longo prazo, ele receberá aproximadamente 20% da CPU. Ao contrário de um agendador de prioridade, onde é muito difícil dizer o que uma prioridade de 40 realmente significa, aqui a regra é clara: um processo que armazena uma fração f dos bilhetes receberá cerca de uma fração f do recurso em questão.

O agendamento por sorteio tem várias propriedades interessantes. Por exemplo, se um novo processo aparece e recebe alguns bilhetes, no próximo sorteio, ele terá uma chance de ganhar em proporção ao número de bilhetes que possui. Em outras palavras, o agendamento por sorteio é altamente responsivo.

Processos cooperativos podem trocar de bilhetes se desejarem. Por exemplo, quando um processo de cliente envia uma mensagem para um processo de servidor e, então, bloqueia, ele pode dar todos os seus bilhetes para o servidor, para aumentar a chance de o servidor executar em seguida. Quando o servidor tiver terminado, ele devolve os bilhetes para que o cliente possa executar novamente. De fato, na ausência de clientes, os servidores não precisam de nenhum bilhete.

O agendamento por sorteio pode ser utilizado para resolver problemas que são difíceis de gerenciar com outros métodos. Um exemplo é um servidor de vídeo em que vários processos estão alimentando seqüências de vídeo para seus clientes, mas a diferentes taxas de quadros. Suponha que os processos precisem de quadros a taxas de 10, 20 e 25 quadros/s. Alocando para esses processos 10, 20 e 25 bilhetes, respectivamente, eles automaticamente dividirão a CPU na proporção correta.

2.4.7 Agendamento de Tempo Real

Um sistema de **tempo real** é um sistema em que o tempo desempenha um papel essencial. Em geral, um ou mais dispositivos físicos externos para o computador geram estímulos, e o computador deve interagir apropriadamente a eles dentro de uma quantidade fixa de tempo. Por exemplo, o computador em um CD *player* recebe os bits à medida que eles vêm da unidade e deve convertê-los em música dentro de um intervalo de tempo muito apertado. Se o cálculo demora muito, a música soará estranha. Outros sistemas de tempo real servem para monitorar pacientes em uma unidade de terapia intensiva de hospital, o piloto automático em uma aeronave e o controle de segurança em um reator nuclear. Em todos esses casos, receber a resposta certa, mas muito tarde é freqüentemente tão ruim quanto não recebê-la.

Os sistemas de tempo real são geralmente classificados como ***hard real time***, que significa que há prazos finais absolutos que devem ser cumpridos, e ***soft real time***, que significa que perder um prazo final ocasionalmente é tolerável. Em ambos os casos, o comportamento de tempo real é alcançado dividindo-se o programa em um número de processos, cujo comportamento é previsível e conhecido de antemão. Tais processos via de regra tem vida curta e podem executar para concluir em menos de um segundo. Quando um evento externo é capturado, é trabalho do agendador agendar os processos de tal maneira que todos os prazos finais sejam cumpridos.

Os eventos que um sistema de tempo real pode ter de responder podem ser classificados mais especificamente como **periódicos** (ocorrendo a intervalos regulares) ou **aperiódicos** (ocorrendo de maneira imprevisível). Um sistema pode ter de responder a múltiplos fluxos periódicos de evento. Dependendo de quanto tempo cada evento requer para processamento, talvez nem seja possível manipular todos eles. Por exemplo, se houver m eventos periódicos e o evento i ocorre com um período P_i e requer C_i segundos de tempo de CPU para gerenciar cada evento, então a carga só pode ser gerenciada se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Um sistema tempo real que satisfaz esse critério é conhecido como **agendável.**

Como exemplo, considere um sistema de *soft real time* com três eventos periódicos, com períodos de 100, 200 e 500ms, respectivamente. Se esses eventos exigirem 50, 30 e 100ms de tempo de CPU por evento, respectivamente, o sistema será agendável porque 0,5 + 0,15 + 0,2 <1. Se um quarto evento com um período de 1 s for adicionado, o sistema permanecerá agendável contanto que esse evento não precise de mais de 150ms de tempo de CPU por ocorrência. Nesse cálculo está implícita a suposição de que a sobrecarga de comutação de contexto é tão pequena que pode ser ignorada.

Algoritmos de agendamento de tempo real podem ser dinâmicos ou estáticos. O primeiro toma suas decisões de agendamento em tempo de execução; o último toma-as antes de o sistema começar a executar. Consideremos brevemente alguns dos algoritmos de agendamento de tempo real dinâmicos. O algoritmo clássico é o **algoritmo de índice monotônico** (Liu e Layland, 1973). De antemão, ele atribui a cada processo uma prioridade proporcional à freqüência de ocorrência do seu evento desencadeado. Por exemplo, um processo que executa a cada 20ms recebe prioridade 50 e um processo que executa a cada 100ms obtém prioridade 10. Em tempo de execução, o agendador sem-

pre executa o processo pronto de mais alta prioridade, fazendo a preempção do processo em execução se necessário. Liu e Layland provaram que esse algoritmo é ótimo.

Outro algoritmo de agendamento de tempo real popular é o de **prazo final mais cedo primeiro.** Sempre que um evento é capturado, seu processo é adicionado à lista de processos prontos. A lista é mantida classificada pelo prazo final, o que para um evento periódico é a próxima ocorrência do evento. O algoritmo executa o primeiro processo na lista, aquele com o prazo final mais próximo.

Um terceiro algoritmo primeiro calcula para cada processo a quantidade de tempo que tem de dispensar, chamada **lassidão.** Se um processo requer 200ms e deve terminar em 250ms, sua lassidão é 50ms. O algoritmo, chamado de **menor lassidão,** escolhe o processo com a menor quantidade de tempo a dispensar.

Enquanto na teoria é possível transformar um sistema operacional de propósito geral em um sistema de tempo real utilizando um desses algoritmos de agendamento, na prática a sobrecarga de comutação de contexto de sistemas de propósito geral é tão grande que o desempenho de tempo real só pode ser alcançado para aplicativos com limitações de tempo folgadas. Como conseqüência, a maior parte do trabalho em tempo real utiliza sistemas operacionais de tempo real especiais que têm certas propriedades importantes. Em geral, essas incluem tamanho pequeno, tempo de interrupção rápido, comutação de contexto rápida, curto intervalo durante o qual as interrupções ficam desativadas e a capacidade de gerenciar múltiplos temporizadores no intervalo de milissegundos ou microssegundos.

2.4.8 Agendamento de Dois Níveis

Até agora vínhamos mais ou menos supondo que todos os processos executáveis estavam na memória principal. Se a memória principal disponível for insuficiente, alguns dos processos executáveis terão de permanecer no disco, inteiros ou em parte. Essa situação tem implicações importantes para o agendamento, uma vez que o tempo de comutação de processo para carregar e executar um processo do disco é de ordens de magnitude superior ao tempo de comutação para um processo já pronto na memória principal.

Uma maneira mais prática de lidar com comutação de processos em disco é utilizar um agendador de dois níveis. Algum subconjunto dos processos executáveis é carregado inicialmente na memória principal, como mostrado na Figura 2-25(a). O agendador, então, restringe-se a escolher processos apenas desse subconjunto temporariamente. Periodicamente, um agendador de nível mais alto é invocado para remover os processos que estiveram por tempo suficiente na memória e carregar processos que estiveram muito tempo em disco. Uma vez que a mudança tenha sido feita, como na Figura 2-25(b), o agendador de nível mais baixo novamente se restringe aos processos em execução que estão realmente na memória. Assim, o agendador de nível mais baixo fica preocupado com fazer uma escolha entre os processos executáveis que estão na memória nesse momento, enquanto o agendador de nível mais alto fica preocupado com o movimento dos processos de um lado a outro entre memória e disco.

Entre os critérios que o agendador de nível mais alto poderia utilizar para tomar suas decisões estão os seguintes:

1. Quanto tempo se passou desde que o processo foi levado para o disco ou para a memória?
2. Quanto tempo de CPU o processo teve recentemente?
3. Qual é o tamanho do processo? (Os pequenos não atrapalham.)
4. Qual é o nível de prioridade do processo?

Aqui, novamente, poderíamos utilizar agendamento por *round robin*, prioridade ou qualquer um dos vários outros métodos. Os dois agendadores podem ou não utilizar o mesmo algoritmo.

2.4.9 Política *versus* Mecanismo

Até agora, vínhamos tacitamente supondo que todos os processos no sistema pertenciam a usuários diferentes e assim estariam competindo pela CPU. Embora isso freqüen-

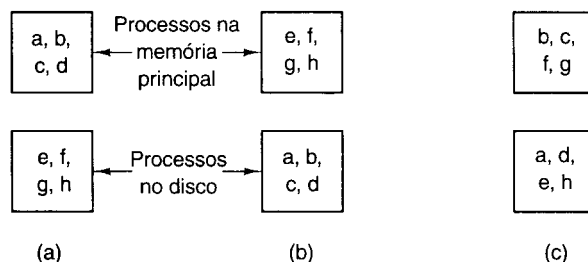



Figura 2-25 Um agendador de dois níveis deve mover processos entre disco e memória e também eleger processos para executar na memória. Três instantes diferentes de tempo são representados por (a), (b) e (c).

temente seja verdadeiro, às vezes acontece de um processo ter muitos filhos que executam sob seu controle. Por exemplo, um processo de sistema de gerenciamento de banco de dados pode ter muitos filhos. Cada filho pode trabalhar em uma requisição diferente, ou cada um pode ter alguma função específica para executar (analisar uma consulta, acessar um disco, etc.). É plenamente possível que o processo principal tenha uma excelente idéia de quais de seus filhos são os mais importantes (ou de tempo crítico) e quais são menos. Infelizmente, nenhum dos agendadores discutidos anteriormente aceita qualquer entrada de processos de usuário sobre decisões de agendamento. Como resultado, o agendador raramente faz a melhor escolha.

A solução para esse problema é separar o **mecanismo de agendamento** da **política de agendamento**. Isso significa que o algoritmo de agendamento é parametrizado de alguma maneira, mas os parâmetros podem ser preenchidos por processos de usuário. Consideremos novamente o exemplo do banco de dados. Suponha que o *kernel* utiliza um algoritmo de agendamento por prioridade, mas fornece uma chamada de sistema por meio da qual um processo pode definir (e alterar) as prioridades de seus filhos. Assim, o pai pode controlar em detalhe como seus filhos são agendados, mesmo que ele não faça o agendamento. Aqui o mecanismo está no *kernel*, mas a política é configurada por um processo de usuário.

2.5 VISÃO GERAL DE PROCESSOS EM MINIX

Tendo completado nosso estudo dos princípios de gerenciamento de processos, comunicação interprocesso e agendamento, agora podemos dar uma olhada em como eles são aplicados no MINIX. Diferentemente do UNIX, cujo *kernel* é um programa monolítico e não dividido em módulos, o MINIX é uma coleção de processos que se comunicam entre si e com processos de usuário utilizando uma única primitiva de comunicação interprocesso —a passagem de mensagem. Esse projeto proporciona uma estrutura mais flexível e modular, tornando fácil, por exemplo, substituir o sistema de arquivos inteiro por um completamente diferente, sem nem mesmo precisar recompilar o *kernel*.

2.5.1 A Estrutura Interna do MINIX

Vamos começar nosso estudo do MINIX com uma breve visão geral do sistema. O MINIX é estruturado em quatro camadas, com cada camada executando uma função bem-definida. As quatro camadas são ilustradas na Figura 2-26.

A camada inferior captura todas as interrupções e *traps**, faz o agendamento e fornece às camadas mais altas um modelo de processos seqüenciais independentes que se comunicam utilizando mensagens. O código nessa camada tem duas funções principais. A primeira é capturar os *traps* e as interrupções, salvar e restaurar registradores, agendar e as demais funções para realmente fazer a abstração de processo oferecida para as camadas mais altas funcionarem. A segunda é gerenciar a mecânica das mensagens; verificar destinos legais, localizar buffers de envio e de recebimento na memória física e copiar bytes do remetente para o destinatário. Essa parte da camada que lida com o nível mais baixo do gerenciamento de interrupções é escrita em linguagem *assembly*. O resto da camada e todas as camadas mais altas são escritos em C.

A camada 2 contém os processos de E/S, um por tipo de dispositivo. Para distingui-los dos processos de usuário normais, iremos chamá-los de **tarefas,** mas as diferenças entre tarefas e processos são mínimas. Em muitos sistemas, as tarefas de E/S são chamadas de ***drivers* de dispositivo;** utilizaremos os termos "tarefa" e "*driver* de dispositivo" intercambiavelmente. Uma tarefa é necessária para cada tipo de dispositivo, incluindo discos, impressoras, terminais, placas de rede e relógios. Se outros dispositivos de E/S estiverem presentes, também é necessária uma tarefa para cada um deles. Uma tarefa, a tarefa de sistema, é um pouco diferente, uma vez que não corresponde a qualquer dispositivo de E/S. Discutiremos as tarefas no próximo capítulo.

Todas as tarefas na camada 2 e todo o código na camada 1 estão vinculados entre si em um único programa binário chamado ***kernel.*** Algumas tarefas compartilham sub-rotinas comuns, mas, em geral, são independentes uma da outra, são agendadas independentemente e comunicam-se utilizando mensagens. Os processadores Intel, desde o 286, atribuem um entre quatro níveis de privilégio a cada processo. Embora as tarefas e o *kernel* sejam compilados juntos, quando o *kernel* e os manipuladores de interrupções estão executando, eles recebem mais privilégios que as tarefas. Portanto, o verdadeiro código do *kernel* pode acessar qualquer parte da memória e qualquer registrador do processador — essencialmente, o *kernel* pode executar qualquer instrução utilizando dados de qualquer lugar no sistema. As tarefas não podem executar todas as instruções no nível de máquina, nem podem acessar todos os registradores da CPU, ou todas as partes da memória. Mas elas podem acessar regiões da memória que pertencem a processos menos privilegiados, para executar uma E/S para elas. Uma tarefa, a tarefa de sistema, não faz E/S no sentido comum mas existe para fornecer serviços, como copiar entre regiões diferentes da memória, para processos que não têm permissão para fazer essas coisas sozinhos. Em máquinas que não oferecem níveis diferentes de privilégio, como processadores Intel mais antigos, tais restrições não podem ser impostas, naturalmente.

A camada 3 contém processos que fornecem serviços úteis para os processos de usuário. Esses processos de servidor executam em um nível menos privilegiado que o *kernel* e as tarefas e não podem acessar portas de E/S direta-

*N. de R. Um *trap* é uma instrução especial que, quando executada pelo processador gera o mesmo efeito de uma interrupção. Um *trap* pode ser visto como uma interrupção ocasionada por software.

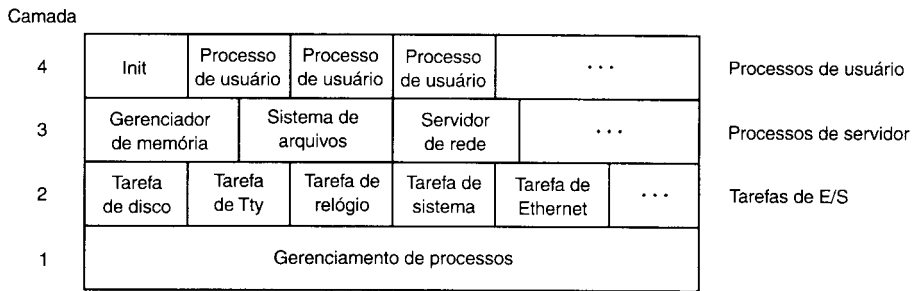



Figura 2-26 O MINIX é estruturado em quatro camadas.

mente. Eles também não podem acessar memória fora dos segmentos atribuídos a eles. O **gerenciador de memória** (MM) executa todas as chamadas de sistema do MINIX que envolvem gerenciamento de memória, como FORK, EXEC e BRK. O **sistema de arquivos** (FS) executa todas as chamadas de sistema de arquivos, como READ, MOUNT e CHDIR.

Como observamos no início do Capítulo 1, os sistemas operacionais fazem duas coisas: gerenciar recursos e fornecer uma máquina estendida implementando chamadas de sistema. No MINIX, o gerenciamento de recursos está em grande parte no *kernel* (camadas 1 e 2) e a interpretação de chamadas de sistema está na camada 3. O sistema de arquivos foi projetado como um "servidor" de arquivos e pode ser movido para uma máquina remota sem quase nenhuma mudança. Isso também se aplica ao gerenciador de memória, embora os servidores de memória remotos não sejam tão úteis quanto servidores de arquivo remotos.

Servidores adicionais também podem existir na camada 3. A Figura 2-26 mostra um servidor de rede ali. Embora o MINIX, como descrito neste livro, não inclua o servidor de rede, seu código-fonte é parte da distribuição padrão do MINIX. O sistema pode facilmente ser recompilado para incluí-lo.

Esse é um bom lugar para observar que embora os servidores sejam processos independentes, eles diferem de processos de usuário por que são iniciados quando o sistema é iniciado e nunca terminam enquanto o sistema está ativo. Adicionalmente, embora executem no mesmo nível de privilégio que os processos de usuário em termos das instruções de máquina que eles têm permissão para executar, eles recebem prioridade mais alta de execução que os processos de usuário. Para acomodar um novo servidor, o *kernel* deve ser recompilado. O código de inicialização do *kernel* instala os servidores em entradas privilegiadas na tabela de processos antes de qualquer processo de usuário obter permissão para executar.

Por fim, a camada 4 contém todos os processos de usuário — os *shells*, editores, compiladores e programas escritos pelo usuário. Um sistema em execução normalmente tem algum processo que é iniciado quando o sistema é inicializado e que executa eternamente. Por exemplo, um **daemon** é um processo de segundo plano que executa periodicamente ou sempre espera algum evento, como a chegada de um pacote de rede. Em certo sentido, um daemon é um servidor que é iniciado independentemente e executa como um processo de usuário. Entretanto, diferentemente dos servidores verdadeiros instalados em entradas privilegiadas, esses programas não podem receber o tratamento especial do *kernel* que os processos servidores de memória e de arquivos recebem.

2.5.2 Gerenciamento de Processos no MINIX

Os processos no MINIX seguem o modelo geral de processo descrito em certa extensão anteriormente neste capítulo. Os processos podem criar subprocessos, que, por sua vez, podem criar mais subprocessos, produzindo uma árvore de processos. De fato, todos os processos de usuário no sistema inteiro são parte de uma única árvore com *init* (veja Figura 2-26) na raiz.

Como essa situação ocorre? Quando o computador é ligado, o hardware lê o primeiro setor da primeira trilha do disco de inicialização para a memória e executa o código que encontra lá. Os detalhes variam dependendo de se o disco de inicialização é um disquete ou um disco rígido. Em um disquete esse setor contém o programa de **inicialização**. Ele é muito pequeno, uma vez que tem de caber em um setor. O programa de inicialização do MINIX carrega um programa maior, o *boot*, que, então, carrega o sistema operacional em si.

Em contraste, os discos rígidos exigem um passo intermediário. Um disco rígido é dividido em **partições**, e o primeiro setor de um disco rígido contém um pequeno programa e a **tabela de partição** do disco. Coletivamente, estes são chamados **registro-mestre de inicialização** (*master boot record*). A parte do programa é executada para ler a tabela de partição e selecionar a partição ativa. A partição ativa tem um programa de inicialização em seu primeiro setor, que é, então, carregado e executado para localizar e para iniciar uma cópia do *boot* na partição, exatamente como acontece quando você inicializa a partir de um disquete.

Em qualquer dos casos, o *boot* procura um arquivo de múltiplas partes no disquete ou na partição e carrega as partes individuais na memória nos locais adequados. As partes incluem o *kernel*, o gerenciador de memória, o sistema de arquivos e *init*, o primeiro processo de usuário. Esse processo de inicialização não é uma operação trivial. As operações que estão nos domínios da tarefa de disco e do sistema de arquivos devem ser executadas por *boot* antes de essas partes do sistema serem ativadas. Em uma seção posterior, retornaremos ao assunto de como o MINIX é iniciado. Por enquanto, basta dizer que uma vez que a operação de carga esteja completa, o *kernel* começa a executar.

Durante sua fase de inicialização, o *kernel* inicia as tarefas e só então o gerenciador de memória, o sistema de arquivos e quaisquer outros servidores que executam na camada 3. Quando todos esses tiverem executados e inicializados, eles bloquearão, esperando algo para fazer. Quando todas tarefas e servidores estiverem bloqueados, *init*, o primeiro processo de usuário, será executado. Ele já está na memória, mas poderia, naturalmente, ser carregado do disco como um programa separado uma vez que tudo mais está funcionando no momento em que ele é iniciado. Entretanto, como *init* é iniciado só nesse momento e nunca é recarregado do disco, é mais fácil apenas incluí-lo no arquivo de imagem de sistema com o *kernel*, tarefas e servidores.

Init inicia lendo o arquivo */etc/ttytab*, o qual lista todos dispositivos terminais potenciais. Esses dispositivos que podem ser utilizados como terminais de *login* (na distribuição-padrão, apenas o console) tem uma entrada no campo *getty* de */etc/ttytab*, e *init* cria um processo-filho para cada um desses terminais. Cada filho normalmente executa */usr/bin/getty*, o qual imprime uma mensagem e depois espera um nome ser digitado. Então */usr/bin/login* é chamado com o nome como seu argumento. Se um terminal particular exigir tratamento especial (p. ex., uma linha discada) */etc/ttytab* pode especificar um comando (como */usr/bin/stty)* para ser executado a fim de iniciar a linha antes de executar *getty*.

Após um *login* bem-sucedido, */bin/login* executa o *shell* do usuário (especificado no arquivo */etc/password* e normalmente */bin/sh* ou */usr/bin/ash*). O *shell* espera comandos serem digitados e, então, cria um novo processo para cada comando. Dessa maneira, os *shells* são os filhos de *init*, os processos de usuário são os netos de *init e* todos os processos de usuário no sistema são parte de uma única árvore.

As duas principais chamadas de sistema do MINIX para gerenciamento de processos são FORK e EXEC. FORK é o único meio de criar um novo processo. EXEC permite criar um processo para executar um programa especificado. Quando um programa é executado, ele recebe uma parte da memória cujo tamanho é especificado no cabeçalho do arquivo de programa. Ele mantém essa quantidade de memória durante toda sua execução, embora a distribuição entre segmento de dados, segmento de pilha e não-utilizado possa variar à medida que o processo executa.

Todas as informações sobre um processo são mantidas na tabela de processos, que é dividida entre o *kernel*, o gerenciador de memória e o sistema de arquivos, com cada um desses tendo os campos que precisa. Quando um novo processo aparece (por FORK), ou um processo antigo termina (por EXIT ou por um sinal), o gerenciador de memória primeiro atualiza sua parte na tabela de processos e, então, envia mensagens para o sistema de arquivos e para o *kernel* informando-os para fazer o mesmo.

2.5.3 Comunicação Interprocesso no MINIX

Três primitivas são fornecidas para enviar e para receber mensagens. Elas são chamadas pelos procedimentos de biblioteca de C

```
send(dest, &message);
```

para enviar uma mensagem ao processo *dest*,

```
receive(source, &message);
```

para receber uma mensagem do processo *source* (ou *QUALQUER*), e

```
send_rec(src_dst, &message);
```

para enviar uma mensagem e esperar uma resposta do mesmo processo. O segundo parâmetro em cada chamada é o endereço local dos dados da mensagem. O mecanismo de passagem de mensagem no *kernel* copia a mensagem do remetente para o destinatário. A resposta (para *send_rec*) sobrescreve a mensagem original. A princípio, esse mecanismo de *kernel* poderia ser substituído por uma função que copia mensagens por uma rede para uma função correspondente em outra máquina, implementando um sistema distribuído. Na prática, isso seria algo complicado pelo fato de o conteúdo dessa mensagem ser, às vezes, ponteiros para estruturas de dados grandes, e um sistema distribuído também teria de providenciar a cópia dos próprios dados pela rede.

Cada processo ou tarefa pode enviar e receber mensagens de processos e de tarefas em sua própria camada e daqueles na camada imediatamente abaixo. Os processos de usuário não podem comunicar-se diretamente com as tarefas de E/S. O sistema impõe essa restrição.

Quando um processo (o que também inclui as tarefas como um caso especial) envia uma mensagem para um processo que atualmente não está esperando uma mensagem, o remetente bloqueia até que o destino faça um RECEIVE. Em outras palavras, o MINIX utiliza o método de *rendez-vous* para evitar os problemas de armazenamento de mensagens enviadas, mas ainda não recebidas. Embora menos flexível que um esquema com armazenamento, ele se mostra adequado para esse sistema, e muito mais simples porque nenhum gerenciamento de buffer é necessário.

2.5.4 Agendamento de Processos no MINIX

O sistema de interrupções é o que mantém um sistema operacional multiprogramado em funcionamento. Os processos bloqueiam quando fazem requisições para entrada, permitindo que outros processos executem. Quando a entrada torna-se disponível, o processo atual em execução é interrompido pelo disco, pelo teclado ou por outro hardware. O relógio também gera interrupções utilizadas para certificar que um processo de usuário em execução que não solicitou entrada acabe abandonando a CPU, para dar a outro processo sua chance de executar. É trabalho da camada mais baixa do MINIX ocultar essas interrupções, transformando-as em mensagens. No que diz respeito aos processos (e tarefas), quando um dispositivo de E/S completa uma operação ele envia uma mensagem para o processo, acordando-o e tornando-o executável.

Cada vez que um processo é interrompido, seja a partir de um dispositivo convencional de E/S ou a partir do relógio, há uma oportunidade para determinar qual processo merece uma oportunidade de executar. Naturalmente, isso também deve ser feito sempre que um processo termina, mas em um sistema como o MINIX as interrupções devidas a operações de E/S ou ao relógio ocorrem mais freqüentemente que o término de um processo. O agendador do MINIX utiliza um sistema de filas em três níveis, correspondentes às camadas 2, 3 e 4 da Figura 2-26. Dentro dos níveis de tarefa e de servidor, os processos executam até bloquearem, mas os processos de usuário são agendados utilizando *round robin*. As tarefas têm a prioridade mais alta, o gerenciador de memória e o servidor de arquivos vêm em seguida e, por último, os processos de usuário.

Ao selecionar um processo para executar, o agendador verifica se qualquer tarefa está pronta. Se uma ou mais estiver pronta, a primeira da fila é executada. Se nenhuma tarefa estiver pronta, um servidor (FS ou MM) é escolhido, se possível; caso contrário um processo do usuário é executado. Se nenhum processo estiver pronto, o processo *IDLE* é escolhido. Esse é um laço que executa até que a próxima interrupção ocorra.

A cada tique do relógio, uma verificação é feita para ver se o processo atual é um processo de usuário que executou mais de 100ms. Se for, o agendador é chamado para ver se outro processo de usuário está esperando a CPU. Se algum for localizado, o processo atual é movido para o fim de sua fila de agendamento e o processo agora no topo é executado. As tarefas, o gerenciador de memória e o sistema de arquivos nunca sofrem preempção pelo relógio, independente de quanto tempo eles tenham estado executando.

2.6 IMPLEMENTAÇÃO DE PROCESSOS EM MINIX

Agora que estamos chegando mais perto do código real, são necessárias algumas palavras sobre a notação que utilizaremos. Os termos "procedimento", "função" e "rotina" serão utilizados intercambiavelmente. Os nomes de variáveis, de procedimentos e de arquivos serão escritos em itálico, como em *rw_flag*. Quando uma variável, procedimento ou nome de arquivo inicia uma frase, ela ou ele é escrito com a primeira letra maiúscula, mas todos os nomes começam com letras minúsculas. As chamadas de sistema estarão em caixa alta, por exemplo, READ.

O livro e o software, ambos os quais estão continuamente desenvolvendo-se, não foram "para as máquinas" no mesmo dia; então, pode haver discrepâncias menores entre as referências ao código, à listagem impressa e à versão do CD-ROM. Essas diferenças, porém, geralmente só afetam uma linha ou duas. O código-fonte impresso no livro também foi simplificado para eliminar código utilizado para compilar opções que não são discutidas no livro.

2.6.1 Organização do Código-Fonte do MINIX

Logicamente, o código-fonte é organizado como dois diretórios. Os caminhos completos para esses diretórios em um sistema MINIX padrão são */usr/include/* e */usr/src/* (uma barra, "/", ao final de um nome de caminho indica referência a um diretório). A localização real dos diretórios pode variar de sistema para sistema, mas normalmente a estrutura dos diretórios abaixo do nível mais alto será a mesma que em qualquer sistema. Neste texto, tais diretórios serão referidos como *include/* e *src/*.

O diretório *include/* contém diversos arquivos de cabeçalho do POSIX padrão. Além disso, ele tem três subdiretórios:

1. *sys/* – este subdiretório contém cabeçalhos adicionais do POSIX.
2. *minix/* – inclui arquivos de cabeçalho utilizados pelo sistema operacional.
3. *ibm/* – inclui arquivos de cabeçalho com definições específicas do IBM PC.

Para suportar extensões para o MINIX e programas que executam no ambiente MINIX, outros arquivos e subdiretórios também estão presentes em *include/* como fornecido no CD-ROM ou na Internet. Por exemplo, o diretório *include/net/* e seu subdiretório *include/net/gen/* suportam extensões de rede. Entretanto, neste texto, apenas os arquivos necessários para compilar o sistema MINIX básico foram impressos e discutidos.

O diretório *grc/* contém três subdiretórios importantes que contêm o código-fonte do sistema operacional:

1. *kernel/* – as camadas 1 e 2 (processos, mensagens e *drivers*).
2. *mm/* – o código para o gerenciador de memória.
3. *fs/* – o código para o sistema de arquivos.

Há três outros diretórios de código-fonte que não são impressos nem discutidos no texto, mas que são essenciais para produzir um sistema funcional:

1. *src/lib/* – o código-fonte para procedimentos de biblioteca (p. ex., *open*, *read*).
2. *src/tools/* – o código-fonte para o programa *init*, utilizado para iniciar o MINIX.
3. *src/boot/* – o código para inicializar e instalar o MINIX.

A distribuição-padrão do MINIX inclui vários outros diretórios de fontes. Um sistema operacional existe, naturalmente, para suportar comandos (programas) que executarão nele. Assim há um grande diretório *src/command* com o código-fonte para os programas utilitários (p. ex., *cat*, *cp*, *date*, *ls*, *pwd*). Como o MINIX é um sistema operacional educacional, destinado a ser modificado, há um diretório *src/test/* com programas projetados para testar completamente um sistema recentemente compilado do MINIX. Por fim, o diretório */src/inet/* inclui o código-fonte para recompilar o MINIX com suporte de rede.

Por conveniência, normalmente iremos referir-nos a nomes simples de arquivo quando estiver claro a partir do contexto qual é o nome completo do caminho. Deve-se notar, entretanto, que alguns nomes de arquivo aparecem em mais de um diretório. Por exemplo, há vários arquivos chamados *const.h* em que constantes relevantes a uma determinada parte do sistema são definidas. Os arquivos em um diretório particular serão discutidos juntos, então, não deve haver nenhuma confusão. Os arquivos são relacionados no Apêndice A na ordem em que eles são discutidos no texto, para facilitar o acompanhamento. A utilização de um par de marcadores de página pode ser útil neste ponto.

Também vale notar que o Apêndice B contém uma lista alfabética de todos os arquivos descritos no Apêndice A; e o Apêndice C contém uma lista de onde localizar as definições de macros, de variáveis globais e de procedimentos utilizados no MINIX.

O código para as camadas 1 e 2 está contido no diretório *src/kernel/*. Neste capítulo, estudaremos os arquivos desse diretório que suportam o gerenciamento de processos, a camada mais baixa da estrutura do MINIX que vimos na Figura 2-26. Essa camada inclui funções que gerenciam a inicialização do sistema, interrupções, passagem de mensagens e agendamento de processos. No Capítulo 3, veremos os demais arquivos deste diretório, que suportam as várias tarefas, a segunda camada na Figura 2-26. No Capítulo 4, examinaremos os arquivos do gerenciador de memória em *src/mm/* e no Capítulo 5, estudaremos o sistema de arquivos, cujos arquivos-fonte estão localizados em *src/fs/*.

Quando o MINIX é compilado, todos os arquivos do código-fonte em *src/kernel/*, *src/mm/* e *src/fs/* são compilados em arquivos-objeto. Todos os arquivos-objeto em *src/kernel/* estão vinculados para formar um único programa executável, *kernel*. Os arquivos-objeto em *src/mm/* também estão vinculados para formar um único programa executável, *mm*. O mesmo se aplica a *fs*. As extensões podem ser aumentadas adicionando-se outros servidores. Por exemplo, suporte de rede é adicionado modificando-se *include/minix/config.h* a fim de ativar a compilação dos arquivos em *src/inet/* para formar *inet*. Outro programa executável, *init*, está dentro de *src/tools/*. O programa *installboot* (cujo fonte está em *src/boot/*) adiciona nomes a cada um desses programas, define que seu comprimento é um múltiplo do tamanho de setor de disco (para tornar mais fácil carregar as partes independentemente) e concatena-os em um único arquivo. Esse novo arquivo é o binário do sistema operacional e pode ser copiado para o diretório-raiz ou o diretório */minix/* de um disquete ou partição de disco rígido. Posteriormente, o programa monitor de inicialização pode carregar e executar o sistema operacional. A Figura 2-27 mostra o arranjo da memória depois que os programas concatenados são separados e carregados. Detalhes, naturalmente, dependem da configuração do sistema. O exemplo na figura é para um sistema MINIX configurado para tirar proveito de um computador equipado com vários megabytes de memória. Isso torna possível alocar um grande número de buffers do sistema de arquivos, mas o grande sistema de arquivos resultante não cabe no intervalo mais baixo da memória, abaixo dos 640K. Se o número de buffers for reduzido drasticamente, é possível fazer todo o sistema caber em menos de 640K de memória, com espaço também para alguns processos de usuário.

É importante saber que o MINIX consiste em três ou mais programas totalmente independentes que se comunicam apenas passando mensagens. Um procedimento chamado *panic* em *src/fs/* não gera conflito com um procedimento chamado *panic* em *src/mm/* porque, em última instância, eles estão vinculados* em arquivos executáveis diferentes. Os únicos procedimentos que as três partes do sistema operacional têm em comum são algumas das rotinas de biblioteca em *lib/*. Essa estrutura modular torna muito fácil modificar, digamos, o sistema de arquivos, sem que essas mudanças afetem o gerenciador de memória. Também torna simples remover o sistema de arquivos inteiro e colocá-lo em uma máquina diferente como um servidor de arquivos, para comunicar-se com máquinas de usuários enviando mensagens por uma rede.

Como outro exemplo da modularidade do MINIX, compilar o sistema com ou sem suporte de rede não faz absolutamente nenhuma diferença para o gerenciador de memória ou para o sistema de arquivos e afeta o *kernel* só porque a tarefa de Ethernet está compilada ali, junto com o suporte para outros dispositivos de E/S. Quando ativado, o servidor de rede é integrado ao sistema MINIX como um servidor com o mesmo nível de prioridade que o gerenciador de memória ou o servidor de arquivos. Sua operação pode envolver a transferência de grandes quantidades de dados muito rapidamente e isso requer prioridade mais alta do que um processo de usuário receberia. Exceto para a tarefa de Ethernet, entretanto, as funções de rede poderiam ser

*N. de R. O termo original é *linked*, de *link*, que, neste contexto, é o processo de gerar um executável a partir de códigos-objeto compilados independentemente.

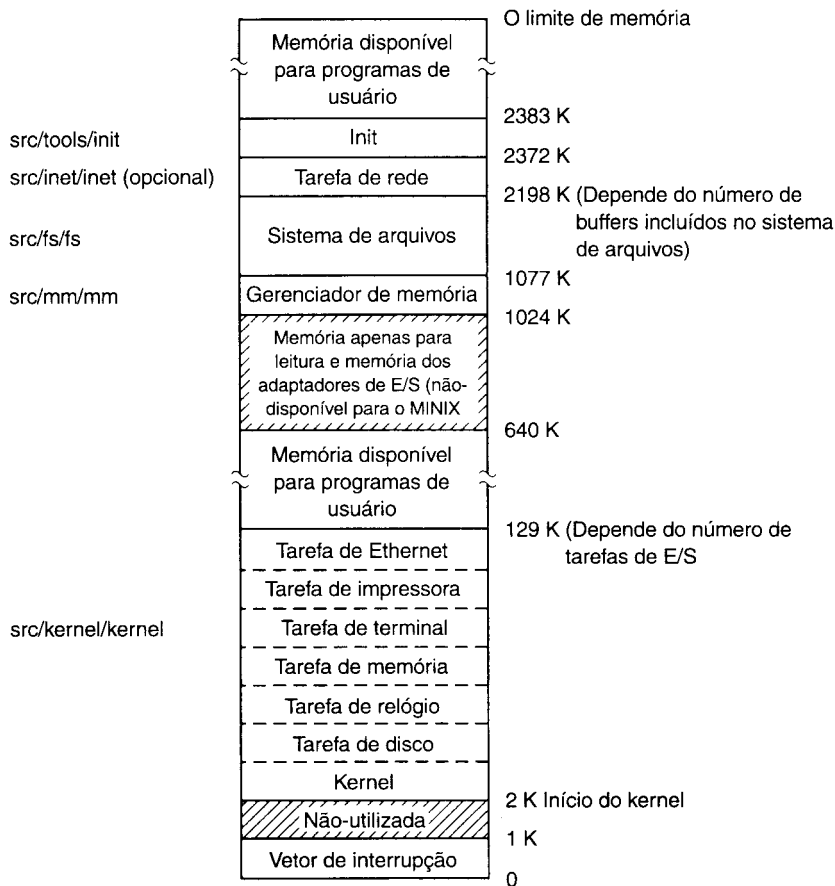



Figura 2-27 O arranjo de memória depois que o MINIX foi carregado a partir do disco para a memória. As quatro (ou cinco, incluindo o suporte de rede) partes compiladas e vinculadas independentemente são bem-distintas. Os tamanhos são aproximados, dependendo da configuração.

executadas por processos no nível do usuário. As funções de rede não são funções tradicionais do sistema operacional, e uma discussão detalhada do código de rede está além do escopo deste livro. Nas próximas seções e capítulos, a discussão será baseada em um sistema MINIX compilado sem suporte de rede.

2.6.2 Os Arquivos de Cabeçalho Comuns

O diretório *include/* e seus subdiretórios contêm uma coleção de arquivos definindo macros, constantes e tipos. O padrão POSIX requer muitas dessas definições e especifica em quais arquivos do diretório principal *include/* e seu subdiretório *include/sys/* será encontrada a definição necessária. Os arquivos nesses diretórios são **arquivos de cabeçalho** (*header files*), identificados pelo sufixo *.h* e utilizados por meio de declarações **#include** em arquivos-fonte em C. Essas declarações são um recurso da linguagem C. Os arquivos de cabeçalho tornam mais fácil a manutenção de um sistema grande.

Os cabeçalhos provavelmente necessários para compilar programas de usuário estão localizados em *include/* enquanto *include/sys/* é tradicionalmente utilizado para arquivos que são utilizados principalmente para compilar programas de sistema e de utilitários. A distinção não é tão importante, e uma compilação típica, seja de um programa de usuário ou de parte do sistema operacional, incluirá arquivos desses dois diretórios. Discutiremos aqui os arquivos que são necessários para compilar o sistema MINIX-padrão, primeiro tratando daqueles em *include/* e, então, daqueles em *include/sys/*. Na próxima seção, discutiremos todos os arquivos nos diretórios *include/minix/* e *include/IBM/*, que, como os nomes de diretório indicam, são únicos para o MINIX e para sua implementação em computadores do tipo IBM.

Os primeiros cabeçalhos a serem considerados são verdadeiramente de propósito geral, tanto que eles não são referenciados diretamente por nenhum dos arquivos-fonte em linguagem C para o sistema MINIX. Em vez disso, eles são incluídos em outros arquivos de cabeçalho, os cabeçalhos-mestres *src/kernel/kernel.h, src/mm/mm.h* e *src/fs/fs.h* para cada uma das três partes principais do sistema MINIX, que, por sua vez, são incluídos em toda compilação. Cada cabeçalho-mestre é padronizado de acordo com as necessidades da parte correspondente do sistema MINIX, mas todos eles iniciam com uma seção como a mostrada na Figura 2-28. Os cabeçalhos-mestres serão discutidos novamente em outras seções do livro. Esta visualização prévia é para enfatizar que os cabeçalhos dos vários diretórios são utilizados juntos. Nesta seção e na próxima, mencionaremos cada um dos arquivos referenciados na Figura 2-28.

Iniciemos com o primeiro cabeçalho em *include/ansi.h* (linha 0000). Este é o segundo cabeçalho que é processado sempre que qualquer parte do sistema MINIX é compilada; só *include/minix/config.h* é processado antes. O propósito de *ansi.h* é testar se o compilador satisfaz os requisitos do Standard C, como definidos pela International Organization for Standards. O Standard C também é chamado ANSI C, desde que o padrão originalmente foi desenvolvido pelo American National Standards Institute (ANSI) antes de ganhar reconhecimento internacional. Um compilador de Standard C define várias macros que, então, podem ser testadas na compilação de programas. *__STDC__* é uma dessas macros e é definida por um compilador padrão para ter o valor de 1, como se o pré-processador de C lesse uma linha como

```
#define __STDC__ 1
```

O compilador distribuído com as versões atuais do MINIX é compatível com o Standard C, mas as versões mais antigas do MINIX foram desenvolvidas antes da adoção do padrão e ainda é possível compilar MINIX com um clássico (Kernighan & Ritchie) compilador de C. A intenção é que o MINIX seja fácil de portar para novas máquinas, e permitir compiladores mais antigos é parte disso. Nas linhas 0023 a 0025, a declaração

```
#define _ANSI
```

é processada se um compilador Standard C estiver em uso. *Ansi.h* define várias macros de diferentes maneiras, dependendo de a macro *_ANSI* ser definida ou não.

A macro mais importante neste arquivo é *_PROTOTYPE*. Essa macro permite escrever protótipos de função na forma

```
_PROTOTYPE (tipo-de-retorno nome-da-função,
(tipo-de-argumento argumento) ...
```

e ter isso transformado pelo pré-processador de C em

```
tipo-de-retorno nome-da-função (tipo-do-argumento
argumento,) ...
```

se o compilador seguir o padrão ANSI C, ou

```
tipo-de-retorno nome-da-função ()
```

se o compilador for antigo (i. e., Kernighan & Ritchie).

Antes de deixarmos *ansi.h*, permita-nos mencionar mais um recurso. O arquivo inteiro está incluído entre as linhas

```
#ifndef _ANSI_H
```

e

```
#endif
```

Na linha imediatamente seguinte ao #ifndef, o *_ANSI_H* em si é definido. Um arquivo de cabeçalho deve ser incluído só uma vez em uma compilação; essa construção assegura que o conteúdo do arquivo será ignorado se ele for incluído várias vezes. Veremos essa técnica utilizada em todos os arquivos de cabeçalho no diretório *include/*.

O segundo arquivo em *include/* que é indiretamente incluído em todos os arquivos-fonte do MINIX é o cabeçalho *limits.h* (linha 0100). Esse arquivo define muitos tamanhos básicos, sejam dos tipos de linguagem como o número de bits em um número inteiro, sejam os limites do sistema operacional como o comprimento do nome de um arquivo. *Errno.h* (linha 0200), também é incluído em todos os cabeçalhos-mestres. Ele contém os números de erro que são retornados para programas de usuário na variável global *errno* quando uma chamada de sistema falha. *Errno* também é utilizado para identificar alguns erros internos, como tentar enviar uma mensagem para uma tarefa

```
#include <minix/config.h>        /* DEVE ser o primeiro */
#include <ansi.h>                /* DEVE ser o segundo */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>
```

Figura 2-28 Parte de um cabeçalho-mestre que assegura a inclusão de arquivos de cabeçalho necessários a todos os arquivos-fonte em C.

inexistente. Os números de erro são negativos para identificá-los como códigos de erro dentro do sistema MINIX, mas devem ser tornados positivos antes de retornar aos programas de usuário. O truque utilizado é que cada código de erro é definido em uma linha como

```
#define EPERM (_SIGN 1)
```

(linha 0236). O cabeçalho-mestre para cada parte do sistema operacional define a macro *_SYSTEM*, mas *_SYSTEM* nunca é definido quando um programa de usuário é compilado. Se *_SYSTEM* é definido, então *_SIGN* é definido como "–"; caso contrário recebe uma definição nula.

O próximo grupo de arquivos a serem considerados não estão incluídos em todos os cabeçalhos-mestres, mas não obstante são utilizados em muitos arquivos-fonte em toda parte no sistema MINIX. O mais importante é *unistd.h* (linha 0400). Esse cabeçalho define muitas constantes, a maioria das quais são requeridas pelo POSIX. Além disso, ele inclui protótipos para muitas funções do C, incluindo todas aquelas utilizadas para acessar chamadas de sistema MINIX. Outro arquivo amplamente utilizado é *string.h* (linha 0600), que fornece protótipos para muitas funções de C utilizadas para gerenciamento de *strings*. O cabeçalho *signal.h* (linha 0700) define os nomes-padrão dos sinais. Também contém protótipos para algumas funções relativas a sinais. Como veremos mais adiante, a manipulação de sinais envolve todas as partes do MINIX.

Fcntl.h (linha 0900) simbolicamente define muitos parâmetros utilizados em operações de controle de arquivo. Por exemplo, permite utilizar a macro *o_rdonly* em vez do valor numérico 0 como um parâmetro para uma chamada *open*. Embora esse arquivo seja referenciado, na maior parte, pelo sistema de arquivos, suas definições também são necessárias em diversos lugares no *kernel* e no gerenciador de memória.

Os demais arquivos em *include/* não são tão amplamente utilizados como os já mencionados. *Stdlib.h* (linha 1000) define tipos, macros e protótipos de função que provavelmente serão necessários na compilação até do mais simples programa em C. É um dos cabeçalhos utilizados mais freqüentemente na compilação de programas de usuário, apesar de no fonte do sistema MINIX ser referenciado apenas por alguns arquivos no *kernel*.

Como veremos quando estudarmos a camada de tarefas no Capítulo 3, o console e a interface de terminal de um sistema operacional são complexos, porque muitos tipos diferentes de hardware têm de interagir com o sistema operacional e com programas de usuário de uma maneira padronizada. O cabeçalho *termios.h* (linha 1100) define constantes, macros e protótipos de função utilizados para controle de dispositivos de E/S tipo terminal. A estrutura mais importante é a estrutura *termios*. Ela contém sinalizadores para indicar vários modos de operação, variáveis para configurar velocidades de transmissão de entrada e de saída e uma matriz para armazenar caracteres especiais, como os caracteres *INTR* e *KILL*. Essa estrutura é requerida pelo POSIX, assim como o são muitas das macros e dos protótipos de função definidos neste arquivo.

Entretanto, para ser tão abrangente como o padrão POSIX foi concebido para ser, ele não fornece tudo que se poderia querer, e a última parte do arquivo, da linha 1241 em diante, fornece extensões para o POSIX. Algumas delas são de valor óbvio, como extensões para definir taxas-padrão de transmissão de dados de 57.600 *baud* e superiores, e suporte para exibição de janelas na tela do terminal. O padrão POSIX não proíbe extensões, já que nenhum padrão razoável poderia incluir tudo. Mas quando se escreve um programa no ambiente MINIX que se destina a ser portável para outros ambientes, alguma cautela é necessária para evitar o uso de definições específicas ao MINIX. Isso é fácil de acontecer. Nesse e em outros arquivos que definem extensões específicas do MINIX, a utilização das extensões é controlada por uma declaração

```
#ifdef _MINIX
```

Se *_MINIX* não for definido, o compilador nem mesmo verá as extensões do MINIX.

O último arquivo que consideraremos em *include/* é *a.out.h* (linha 1400), um cabeçalho que define o formato dos arquivos em que os programas executáveis são armazenados em disco, incluindo a estrutura de cabeçalho utilizada para iniciar a execução de um arquivo e a estrutura da tabela de símbolos produzida pelo compilador. Ele é referenciado só pelo sistema de arquivos.

Agora vamos prosseguir para o subdiretório *include/sys/*. Como mostrado na Figura 2-28, todos os cabeçalhos-mestres para as principais partes do sistema MINIX incluem *sys/types.h* (linha 1600) imediatamente após ler *ansi.h*. Esse cabeçalho define muitos tipos de dados utilizados pelo MINIX. Os erros que poderiam surgir da má interpretação de quais tipos de dados fundamentais são utilizados em uma situação particular podem ser evitados, utilizando-se as definições fornecidas aqui. A Figura 2-29 mostra o modo como os tamanhos, em bits, de alguns tipos definidos neste arquivo diferem quando compilados para processadores de 16 bits ou de 32 bits. Note que todos os nomes de tipo terminam com "_t". Isso não é só uma convenção; é um requisito do padrão POSIX. É um exemplo de um **sufixo reservado** e não deve ser utilizado como um sufixo de qualquer nome que *não* seja um nome de tipo.

Embora não seja tão amplamente utilizado para ser incluído nos cabeçalhos-mestres de cada seção, *sys/ioctl.h* (linha 1800) define muitas macros utilizadas para operações de controle de dispositivo. Ele também contém o protótipo para a chamada de sistema IOCTL. Essa chamada não é diretamente invocada por programadores em muitos casos, uma vez que as funções definidas para o POSIX e as prototipadas em *include/termios.h* substituíram muitos antigos usos da antiga função de biblioteca *ioctl* para lidar com terminais, com consoles e com dispositivos semelhantes. Mas ela ainda é necessária. De fato, as funções do POSIX para controle de dispositivos terminais são con-

Tipo	MINIX de 16 bits	MINIX de 32 bits
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figura 2-29. O tamanho, em bits, de alguns tipos em sistemas de 16 bits e de 32 bits.

vertidas em chamadas de sistema IOCTL pela biblioteca. Além disso, há um número sempre crescente de dispositivos, todos os quais precisam de vários tipos de controle, que podem ser interfaceados com um moderno sistema de computador. Por exemplo, perto do fim deste arquivo estão várias definições de códigos de operação que começam com *DSPIO*, para controlar um processador digital de sinais. De fato, a diferença principal entre o MINIX como descrito neste livro e outras versões, é que para os propósitos do livro descrevemos um MINIX com relativamente poucos dispositivos de entrada/saída. Muitos outros, como interfaces de rede, unidades de CD-ROM e placas de som, podem ser adicionados; códigos de controle para todos esses são definidos como macros neste arquivo.

Vários outros arquivos nesse diretório são amplamente utilizados no sistema do MINIX. O arquivo *sys/sigcontext.h* (linha 2000) define estruturas para conservar e para restaurar a operação normal do sistema antes e depois da execução de uma rotina de manipulação de sinal e é utilizado tanto no *kernel* como no gerenciador de memória. Há suporte no MINIX para monitorar executáveis e analisar *dumps* de núcleo com um programa depurador, e *sys/ptrace.h* (linha 2200) define as várias operações possíveis com a chamada de sistema PTRACE. *Sys/stat.h* (linha 2300) define a estrutura que vimos na Figura 1-12, retornada pelas chamadas de sistema STAT e FSTAT, assim como os protótipos das funções *stat* e *fstat* e outras funções utilizadas para manipular propriedades de arquivos. Ele é referenciado em várias partes do sistema de arquivos e do gerenciador de memória.

Os últimos dois arquivos que discutiremos nesta seção não são tão amplamente referenciados como os discutidos acima. *Sys/dir.h* (linha 2400) define a estrutura de uma entrada de diretório do MINIX. É diretamente referenciado apenas uma vez, mas essa referência o inclui em outro cabeçalho que é amplamente utilizado no sistema de arquivos. É importante porque, entre outras coisas, informa quantos caracteres um nome de arquivo pode conter. Por fim, o cabeçalho *sys/wait.h* (linha 2500) define macros utilizadas pelas chamadas de sistema WAIT e WAITPID, que são implementadas no gerenciador de memória.

2.6.3 Arquivo de Cabeçalhos do MINIX

Os subdiretórios *include/minix/* e *include/ibm/* contêm arquivos de cabeçalho específicos do MINIX. Os arquivos em *include/minix/* são necessários para uma implementação do MINIX em qualquer plataforma, embora haja definições alternativas específicas de plataforma dentro de alguns deles. Os arquivos em *include/ibm/* definem estruturas e macros que são específicas do MINIX quando implementado em máquinas tipo IBM.

Iniciaremos com o diretório *minix/*. Na seção anterior, foi visto que *config.h* (linha 2600) é incluído nos cabeçalhos-mestres por toda parte no sistema MINIX e, portanto, ele é o primeiro arquivo realmente processado pelo compilador. Em muitas ocasiões, quando diferenças no hardware ou no modo como o sistema operacional destina-se a ser utilizado requerem mudanças na configuração do MINIX. Editar esse arquivo e recompilar o sistema é tudo o que deve ser feito. Todos os parâmetros configuráveis pelo usuário estão na primeira parte do arquivo. O primeiro desses é o parâmetro *MACHINE*, que pode assumir valores como *IBM_PC*, *SUN_4*, *MACINTOSH* ou outro valor, dependendo do tipo de máquina para o qual o MINIX está sendo compilado. A maior parte do código para o MINIX é independente do tipo de máquina, mas um sistema operacional sempre tem algum código dependente do sistema. Nos poucos lugares neste livro onde discutimos código que é escrito de maneira diferente para sistemas distintos, utilizaremos como exemplos o código para máquinas IBM PC com microprocessadores avançados (80386, 80486, Pentium, Pentium Pro) que utilizam palavras de 32 bits. Todos esses serão referidos como processadores Intel de 32 bits. O MINIX também pode ser compilado para IBM-PCs mais antigos com um tamanho de palavra de 16 bits, e as partes do MINIX dependentes da máquina devem ser codificadas diferentemente para essas máquinas. Em um PC, o próprio compilador determina o tipo de máquina para a qual o MINIX será compilado. O compilador MINIX padrão para PC é o compilador Amsterdam Compiler Kit (ACK). Ele se identifica definindo, além da *macro* _ *_STDC_* _, a macro _ *_ACK_* _. Ele também define uma macro, *_EM_WSIZE*,

que é o tamanho de palavra (em bytes) para sua máquina de destino. Nas linhas 2626 a 2628, o valor de *_EM_WSIZE* é atribuído a macro *_WORD_SIZE*. Mais adiante no arquivo e em várias partes de outros arquivos-fonte MINIX, essas definições são utilizadas. Por exemplo, as linhas 2647 a 2650 começam com o teste

```
#if (MACHINE == IBM_PC && _WORD_SIZE == 4)
```

e definem um tamanho para o *cache* do sistema de arquivos em sistemas de 32 bits.

Outras definições em *config.h* permitem a personalização para outras necessidades em uma instalação particular. Por exemplo, há uma seção que permite que vários tipos de *drivers* de dispositivo sejam incluídos quando o *kernel* do MINIX é compilado. Essa é provavelmente a parte mais freqüentemente editada do código-fonte do MINIX. Essa seção inicia com:

```
#define ENABLE_NETWORKING 0
#define ENABLE_AT_WINI 1
#define ENABLE_BIOS_WINI 0
```

Mudando o 0 na primeira linha para 1 podemos compilar um *kernel* do MINIX para uma máquina que precisa suporte de rede. Definindo *ENABLE_AT_WINI* como 0 e *ENABLE_BIOS_WINI* como 1, podemos eliminar o código do acionador de disco rígido tipo AT (i. e., IDE) e utilizar o BIOS do PC para suporte de disco rígido.

O próximo arquivo é *const.h* (linha 2900), que ilustra outra utilização comum de arquivos de cabeçalho. Aqui encontraremos uma variedade de definições de constantes que provavelmente não serão alteradas quando se compilar um novo *kernel*, mas que são utilizadas em vários lugares. A definição delas aqui ajuda a prevenir erros que poderiam ser difíceis de monitorar se definições inconsistentes fossem feitas em múltiplos lugares. Há outros arquivos chamados *const.h* na árvore de fontes do MINIX, mas eles são de uso limitado. As definições utilizadas somente no *kernel* são incluídas em *src/kernel/const.h*. As definições utilizadas apenas no sistema de arquivos são incluídas em *src/fs/const.h*. O gerenciador de memória utiliza *src/mm/const.h* para suas definições locais. Só as definições que são utilizadas em mais de uma parte do sistema MINIX são incluídas em *include/minix/const.h*.

Algumas das definições em *const.h* são dignas de nota. *EXTERN* é definida como uma macro que se expande em *extern* (linha 2906). As variáveis globais que são declaradas em arquivos de cabeçalho e incluídas em dois ou mais arquivos são declaradas *EXTERN*, como em

```
EXTERN int who;
```

Se a variável for declarada apenas como

```
int who;
```

e incluída em dois ou mais arquivos, alguns *linkeditors* reclamariam de uma variável multiplamente definida. Além disso, o manual de referência de C (Kernighan e Ritchie, 1988) explicitamente proíbe tal construção.

Para evitar esse problema, é necessário ter a declaração assim:

```
extern int who;
```

em todos os lugares, exceto um. Utilizando *EXTERN*, evita-se esse problema porque ela se expande em *extern* em todo lugar que *const.h* é incluída, exceto após uma redefinição explícita de *EXTERN* como uma *string* nula. Isso é feito em cada parte do MINIX, colocando-se definições globais em um arquivo especial chamado *glo.h*, por exemplo, *src/kernel/glo.h*, que indiretamente é incluído em cada compilação. Dentro de cada *glo.h* há uma seqüência

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

e nos arquivos *table.c* de cada parte do MINIX há uma linha

```
#define _TABLE
```

precedendo a seção **#include**. Assim, quando os arquivos de cabeçalho são incluídos e expandidos como parte da compilação de *table.c*, *extern* não é inserida em qualquer lugar (porque *EXTERN* é definida como uma *string* nula dentro de *table.c*) e o armazenamento para as variáveis globais é reservado apenas em um lugar, no arquivo objeto *table.o*.

Se você é iniciante em programação em C e não entende bem o que está ocorrendo aqui, não se apavore; os detalhes realmente não são importantes. A inclusão de múltiplos arquivos de cabeçalho pode causar problemas para alguns *linkeditors* porque pode levar à múltiplas declarações para variáveis incluídas. O uso de *EXTERN* é simplesmente uma maneira de tornar o MINIX mais portável para que ele possa ser vinculado em máquinas cujos *linkeditors* não aceitam variáveis multiplamente definidas.

PRIVATE é definido como um sinônimo de *static*. Os procedimentos e os dados que não são referenciados fora dos arquivos nos quais sempre são declarados como *PRIVATE* para evitar que seus nomes sejam visíveis de fora desses arquivos. Como regra geral, todas as variáveis e procedimentos devem ser declarados como de escopo local sempre que possível. *PUBLIC* é definida como *string* nula. Assim, a declaração

```
PUBLIC void free_zone(Dev_t dev, zone_t numb)
```

sai do pré-processador do C como

```
void free_zone(Dev_t dev, zone_t numb)
```

que, de acordo com as regras de escopo de C, significa que o nome *free_zone* é exportado do arquivo e pode ser utilizado em outros arquivos. *PRIVATE* e *PUBLIC* não são necessários, mas são tentativas de desfazer o dano causado pelas regras de escopo do C (o padrão é os nomes serem exportados para fora do arquivo; deveria ser exatamente o inverso).

O resto de *const.h* define constantes numéricas utilizadas por todo o sistema. Uma seção de *const.h* é dedicada a definições de máquina ou dependentes de configuração. Por exemplo, por todo o código-fonte, a unidade básica de tamanho de memória é o clique. O tamanho de um clique depende da arquitetura do processador, e as alternativas para arquiteturas Intel, Motorola 68000 e Sun SPARC são definidas nas linhas 2957 a 2965. Esse arquivo também contém as macros *MAX* e *MIN*, assim podemos dizer

```
z = MAX (x, y);
```

para atribuir o maior de *x* e *y* a *z*.

Type.h (linha 3100) é outro arquivo incluído em toda compilação por meio dos cabeçalhos-mestres. Ele contém um número de definições de tipos-chave, junto com os valores numéricos relacionados. A definição mais importante nesse arquivo é *message* nas linhas 3135 a 3146. Embora pudéssemos ter definido *message* como sendo uma matriz de algum número de bytes, é melhor como prática de programação tê-la como uma estrutura contendo a união dos vários tipos de mensagem possíveis. Seis formatos de mensagem, *mess_1* a *mess_6*, são definidos. Uma mensagem é uma estrutura que contém um campo *m_source*, informando quem enviou a mensagem, um campo *m_type*, informando qual é o tipo da mensagem (p. ex., GET_TIME para a tarefa de relógio) e os campos de dados. Os seis tipos de mensagem são mostrados na Figura 2-30, na qual, o primeiro e o segundo tipos de mensagem parecem idênticos, assim como o quarto e o sexto tipos. Isso é verdadeiro para o MINIX quando implementado em uma CPU Intel com um tamanho de palavra de 32 bits, mas não seria o caso em uma máquina onde os tipos *int* e *longs* e ponteiros fossem de tamanhos diferentes. Definir seis formatos distintos torna mais fácil recompilar para uma arquitetura diferente.

Quando é necessário enviar uma mensagem que contém, digamos, três inteiros e três ponteiros (ou três inteiros e dois ponteiros), então, o primeiro formato na Figura 2-30 é utilizado. O mesmo se aplica aos outros formatos. Como se atribui um valor ao primeiro número inteiro no primeiro formato? Suponha que a mensagem chame-se *x*. Então, *x.m_u* refere-se à parte da união na estrutura da mensagem. Para referir-se à primeira das seis alternativas na união, utilizamos *x.m_u.m_m1*. Por fim, para chegar ao primeiro número inteiro nessa estrutura, usamos *x.m_u.m_m1.m1i1*. Isso é bem-verboso, assim nomes de campo um pouco mais curtos são definidos como macros depois da definição de *message* em si. Assim *x.m1_i1* pode ser utilizado em vez de *x.m_u.m_m1*. Todos nomes curtos têm a forma da letra *m*, o número do formato, um sublinhado, uma ou duas letras que indicam se o campo é um inteiro, um ponteiro, um longo, um caractere, uma matriz de caracteres ou uma função e um número de seqüência para distinguir múltiplas instâncias do mesmo tipo dentro de uma mensagem.

A propósito, quando discutimos formatos de mensagem, é uma boa oportunidade para observar que um sistema operacional e seu compilador freqüentemente têm um "entendimento" sobre coisas como o arranjo das estruturas e isso pode tornar a vida do implementador mais fácil. No MINIX, os campos *int* em mensagens são, às vezes, utilizados para armazenar tipos de dados *unsigned*. Em alguns casos, isso poderia causar *overflow*, mas o código foi escrito utilizando o conhecimento de que o compilador do MINIX mapeia tipos *unsigned* para *ints* e vice-versa sem alterar os dados ou gerar código para detectar o *overflow*. Uma abordagem mais compulsiva seria substituir cada campo *int* por uma união de *int* e *unsigned*. O mesmo se aplica aos campos *long* nas mensagens; alguns deles podem ser utilizados para passar dados *unsigned long*. Estamos trapaceando aqui? Talvez, alguém poderia dizer, mas se você deseja portar o MINIX para uma nova plataforma, evidentemente, o formato exato das mensagens é algo em que você deve prestar muita atenção; e agora considere-se alertado sobre o fato de que o comportamento do compilador é outro fator que precisa de atenção.

Há um outro arquivo em *include/minix* que é universalmente utilizado, por meio da inclusão nos cabeçalhos mestres. Trata-se de *syslib.h* (linha 3300), que contém protótipos para as funções de biblioteca de C chamadas a partir de dentro do sistema operacional para acessar outros serviços do sistema operacional. As bibliotecas de C não são discutidas em detalhe neste texto, mas muitas delas são padrão e estarão disponíveis em qualquer compilador C. Entretanto, as funções do C referenciadas por *syslib.h* são naturalmente bem-específicas para o MINIX, e portar o MINIX para um novo sistema com um compilador diferente requer portar a maioria dessas funções de biblioteca. Felizmente, isso não é difícil, uma vez que essas funções simplesmente extraem os parâmetros da chamada de função e os inserem em uma estrutura de mensagem, depois enviam a mensagem e extraem os resultados da mensagem de resposta. Muitas dessas funções de biblioteca são definidas em uma dúzia ou menos de linhas de código em C.

Quando um processo precisa executar uma chamada de sistema do MINIX, ele envia uma mensagem para o gerenciador de memória (MM para abreviar) ou o sistema de arquivos (FS para abreviar). Cada mensagem contém o número da chamada de sistema desejada. Esses números são definidos no arquivo seguinte, *callnr.h* (linha 3400).

O arquivo *com.h* (linha 3500) contém, principalmente, definições comuns utilizadas em mensagens do FS e do MM para as tarefas de E/S. Os números de tarefa também são definidos. Para diferenciar dos números de processo, os números de tarefa são negativos. Esse cabeçalho também define os tipos de mensagem (códigos de função) que podem ser enviados para cada tarefa. Por exemplo, a tarefa de relógio aceita códigos *SET_ALARM* (que é utilizado para configurar um temporizador), *CLOCK_TICK* (quando uma interrupção de relógio ocorreu), *GET_TIME* (requisição da

m_source	m_source	m_source	m_source	m_source	m_source
m_type	m_type	m_type	m_type	m_type	m_type
m1_i1	m2_i1	m3_i1	m4_l1	m5_c2 m5_c1	m6_i1
m1_i2	m2_i2	m3_i2	m4_l2	m5_i1	m6_i2
m1_i3	m2_i3	m3_p1	m4_l3	m5_i2	m6_i3
m1_p1	m2_l1	m3_ca1	m4_l4	m5_l1	m6_l1
m1_p2	m2_l2		m4_l5	m5_l2	m6_f1
m1_p3	m2_p1			m5_l3	

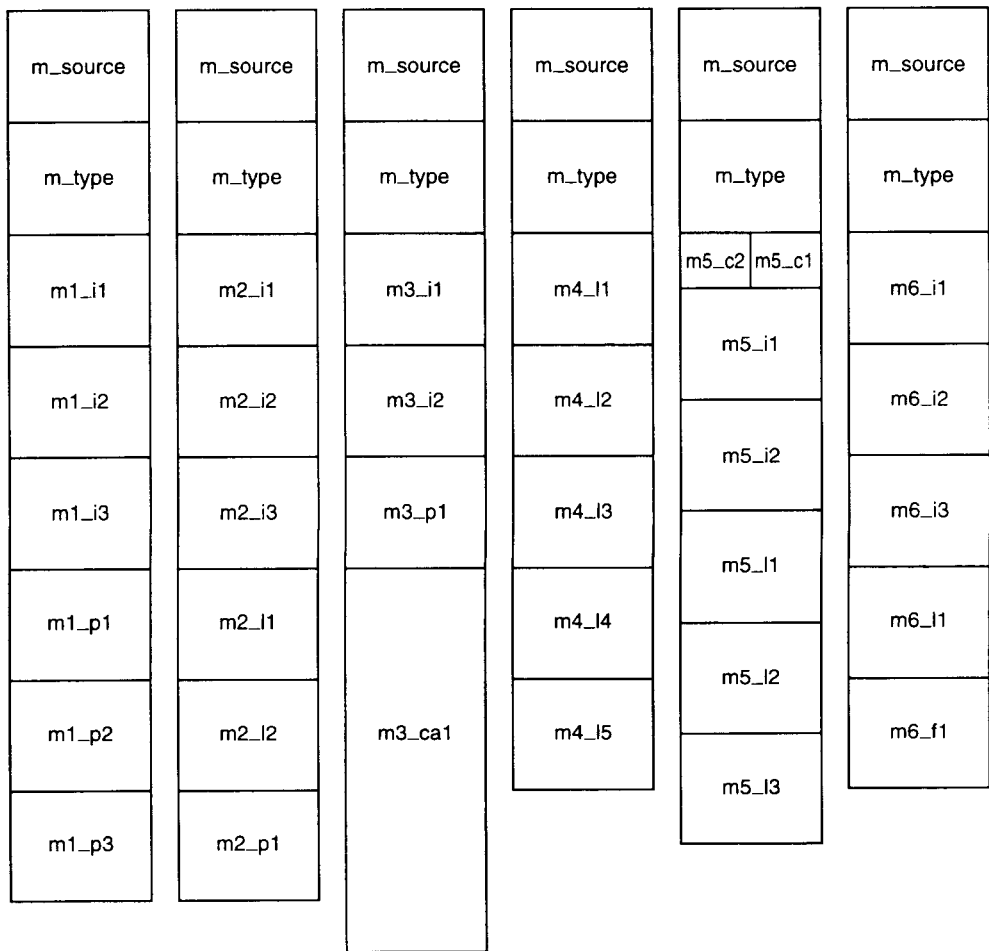



Figura 2-30 Os seis tipos de mensagens utilizados no MINIX. O tamanho dos elementos de uma mensagem irão variar, dependendo da arquitetura da máquina: esse diagrama ilustra tamanhos em uma máquina com ponteiros de 32 bits, como o Pentium (Pro).

hora atual) e *SET_TIME* (para configurar a hora atual). O valor *REAL_TIME* é o tipo de mensagem de resposta para a requisição *GET_TIME*.

Por fim, *include/minix/* contém vários cabeçalhos mais especializados. Entre esses estão *boot.h* (linha 3700), que é utilizado tanto pelo *kernel* como pelo sistema de arquivos para definir dispositivos e acessar parâmetros passados para o sistema pelo programa *boot*. Outro exemplo é *keymap.h* (linha 3800), que define as estruturas para implementar leiautes especializados de teclado para os conjuntos de caractere requeridos por idiomas diferentes. Também é necessário para programas que geram e carregam essas tabelas. Alguns arquivos aqui, como *partition.h* (linha 4000), são utilizados só pelo *kernel* e não pelo sistema de arquivos nem pelo gerenciador de memória. Em uma implementação com suporte para dispositivos adicionais de E/S, há mais arquivos de cabeçalho, para suportar outros dispositivos. Sua colocação nesse diretório precisa de explicação. Idealmente, todos os programas de usuário acessariam dispositivos só pelo sistema operacional, e arquivos como esse seriam colocados em *src/kernel/*. Entretanto, a realidade do gerenciamento de sistema requer que haja alguns comandos de usuário que acessem as estruturas no nível de sistema, como os comandos para fazer partições de disco. É para suportar esses programas utilitários que esses arquivos de cabeçalho especializado são colocados na árvore de diretório *include/*.

O último diretório de cabeçalho especializado que consideraremos, *include/ibm/*, contém dois arquivos que fornecem definições relacionadas com família de computadores IBM PC. Um desses é *diskparm.h*, que é necessário pela tarefa de disquete. Embora essa tarefa seja incluída

na versão-padrão do MINIX, seu código-fonte não é discutido em detalhe nesse texto, uma vez que ele é muito semelhante à tarefa de disco rígido. O outro arquivo nesse diretório é *partition.h* (linha 4100), que define tabelas de partição de disco e constantes relacionadas utilizadas em sistemas compatíveis com IBM-PC. Essas estão colocadas aqui para facilitar portar o MINIX para outra plataforma de hardware. Para hardware diferente, *include/ibm/partition.h* teria de ser substituído, presumivelmente por um *partition.h* em outro diretório apropriadamente nomeado, mas a estrutura definida no arquivo *include/minix/partition.h* é interna ao MINIX e deve permanecer inalterada em um MINIX hospedado em uma plataforma de hardware diferente.

2.6.4 Estruturas de Dados de Processo e Arquivos de Cabeçalho

Agora vamos aprofundar-nos e ver o que significa o código em *src/kernel/*. Nas duas seções anteriores, estruturamos nossa discussão em torno de um trecho de um cabeçalho-mestre típico; primeiro veremos o cabeçalho-mestre real para o *kernel*, *kernel.h* (linha 4200). Ele começa definindo três macros. A primeira, *_POSIX_SOURCE* é uma **macro de teste de recurso** definida pelo próprio padrão POSIX. Exige-se que todas essas macros comecem com o caractere de sublinhado, "_". O efeito de definir a macro *_POSIX_SOURCE* é assegurar que todos os símbolos necessários pelo padrão e quaisquer outros explicitamente permitidos, mas não requeridos, sejam visíveis, enquanto oculta quaisquer símbolos adicionais que são extensões não-oficiais ao POSIX. Já mencionamos as próximas duas definições: a macro *_MINIX* anula o efeito de *_POSIX_SOURCE* para extensões definidas pelo MINIX, e *_SYSTEM* poder ser testada onde quer que seja importante fazer algo diferente ao compilar o código do sistema, em oposição ao código de usuário, como mudar o sinal dos códigos de erro. *Kernel.h*, então, inclui outros arquivos de cabeçalho a partir de *include/* e seus subdiretórios *include/sys/* e *include/minix/*, incluindo todos aqueles referenciados na Figura 2-28. Discutimos todos esses arquivos nas duas seções anteriores. Por fim, quatro outros cabeçalhos do diretório local, *src/kernel/*, são incluídos.

Esse é um bom lugar para apontar para os iniciantes em linguagem C como os nomes de arquivo são citados em um declaração **#include**. Cada compilador C tem um diretório-padrão em que procura para incluir arquivos. Normalmente, este é */usr/include/*, como ocorre em um sistema MINIX padrão. Quando o nome de um arquivo a ser incluído é citado entre os símbolos 'menor que' e 'maior que' ("< ... >"), o compilador procura o arquivo no diretório include-padrão ou em um subdiretório específico do padrão. Quando o nome é citado entre aspas normais (" "..." "), o arquivo é procurado primeiro no diretório atual (ou em um subdiretório especificado) e, então, se não estiver localizado ali, no diretório padrão.

Kernel.h torna possível garantir que todos os arquivos-fonte compartilhem um grande número de definições importantes escrevendo a simples linha

```
#include "kernel.h"
```

em cada um dos outros arquivos-fonte do *kernel*. Como, às vezes, é importante a ordem de inclusão dos arquivos de cabeçalho, *kernel.h* também assegura que essa ordem seja feita corretamente uma vez e para sempre. Isso carrega para um nível mais alto a técnica "faça certo uma vez e, então, esqueça os detalhes" incorporada no conceito de arquivo de cabeçalho. Há cabeçalhos-mestres semelhantes nos diretórios de fontes do sistema de arquivos e do gerenciador de memória.

Agora passemos a examinar os quatro arquivos de cabeçalho locais incluídos em *kernel.h*. Assim como temos arquivos *const.h* e *type.h* no diretório de cabeçalho comum *include/minix/*, também temos arquivos *const.h*. e *type.h* no diretório-fonte do *kernel*, *src/kernel/*. Os arquivos em *include/minix/* estão colocados aí porque são necessários para muitas partes do sistema, incluindo programas que executam sob o controle do sistema. Os arquivos em *src/kernel/* fornecem definições necessárias só para a compilação do *kernel*. Os diretórios-fonte do FS e do MM também contêm arquivos *const.h* e *type.h* para definir constantes e tipos necessários só para essas partes do sistema. Os outros dois arquivos incluídos no cabeçalho-mestre, *proto.h* e *glo.h*, não têm correspondentes nos diretórios principais de *include/*, mas veremos que eles, também, têm correspondentes utilizados na compilação do sistema de arquivos e do gerenciador de memória.

Const.h (linha 4300) contém alguns valores dependentes de máquina, isto é, valores que se aplicam aos chips de CPU da Intel, mas que provavelmente são diferentes quando o UNIX é compilado para hardware diferente. Esses valores são incluídos entre as declarações

```
#if (CHIP == INTEL)
```

e

```
#endif
```

(linhas 4302 até 4396) para agrupá-los.

Ao compilar o MINIX para um chip Intel as macros *CHIP* e *INTEL* são definidas e configuradas iguais em *include/minix/config.h* (linha 2768), e assim o código dependente de máquina será compilado. Quando o MINIX foi portado para um sistema baseado no Motorola 68000, as pessoas que fizeram essa portagem adicionaram seções de código agrupadas por

```
#if (CHIP == M68000)
```

e

```
#endif
```

e fizeram mudanças apropriadas em *include/minix/config.h* para que a linha

```
#define CHIP M68000
```

fosse efetiva. Dessa maneira, o MINIX pode lidar com código e constantes que são específicos de um sistema. Essa construção não aumenta especialmente a legibilidade, então, deve ser utilizada o mínimo possível. De fato, em favor da legibilidade, removemos muitas seções do código dependentes de máquina, para o 68000 e para outros processadores da versão do código impresso neste texto. O código distribuído no CD-ROM e pela Internet mantém o suporte para outras plataformas.

Algumas das definições em *const.h* merecem menção especial. Algumas delas são dependentes de máquina, como importantes vetores de interrupção e de valores de campo utilizados para reinicializar o chip controlador de interrupções após cada interrupção. Cada tarefa dentro do *kernel* tem sua própria pilha, mas no gerenciamento de interrupções é utilizada uma pilha especial do tamanho de *K_STACK_BYTES*, definido aqui na linha 4304. Isso também é definido dentro da seção dependente de máquina, uma vez que uma arquitetura diferente poderia requerer mais ou menos espaço de pilha.

Outras definições são independentes de máquina, mas necessárias para muitas partes do código do *kernel*. Por exemplo, o agendador do MINIX tem *NQ* (3) filas de prioridade, chamadas *TASK_Q* (prioridade alta), *SERVER_Q* (prioridade média) e *USER _Q* (prioridade baixa). Os nomes são utilizados para fazer o código-fonte compreensível, mas é o valor numérico definido por essas macros que realmente é compilado no programa executável. Por fim, a última linha de *const.h* define *printf* como uma macro que será traduzida como *printk*. Isso permite que o *kernel* imprima mensagens no console, como mensagens de erro, utilizando um procedimento definido dentro do *kernel*. Esse é um desvio do mecanismo normal, que requer passar mensagens do *kernel* para o sistema de arquivos, e então do sistema de arquivos para a tarefa de impressora. Durante uma falha de sistema isso pode não funcionar. Veremos chamadas para *printf*; aliás *printk*, em uma chamada de procedimento do *kernel* chamada *panic* que, como se poderia esperar, é invocada quando erros fatais são detectados.

O arquivo *type.h* (linha 4500) define vários protótipos e estruturas utilizadas em qualquer implementação do MINIX. A estrutura *tasktab* define a estrutura de um elemento da matriz *tasktab*, e a estrutura *memory* (linhas 4513 a 4516) define as duas quantidades que singularmente especificam uma área da memória. Eis um bom lugar para mencionar alguns conceitos utilizados em referência à memória. Um *click* é a unidade básica de medida de memória; no MINIX para processadores Intel, um *click* equivale a 256 bytes. A memória é medida como *phys_clicks*, que pode ser utilizada pelo *kernel* para acessar qualquer elemento da memória em qualquer lugar no sistema, ou como *vir_clicks*, utilizada por processos outros que não o *kernel*. Uma referência de memória *vir_clicks* é sempre com relação à base de um segmento de memória atribuído a um processo particular, e o *kernel* freqüentemente tem de fazer traduções entre os dois. A inconveniência disso é compensada pelo fato de que um processo pode fazer todas as suas próprias referências de memória em *vir_clicks*. Alguém poderia supor que a mesma unidade poderia ser utilizada para especificar o tamanho de qualquer tipo de memória, mas há uma vantagem em utilizar *vir_clicks* para especificar o tamanho de uma unidade de memória atribuída a um processo, uma vez que, quando essa unidade é utilizada, uma verificação é feita para assegurar que nenhuma memória seja acessada fora do que foi especificamente atribuído ao processo atual. Esse é um recurso importante do **modo protegido** dos processadores Intel modernos, como o Pentium e o Pentium Pro. Sua ausência nos processadores anteriores 8086 e 8088 causou algumas dores de cabeça no projeto das primeiras versões do MINIX.

Type.h também contém várias definições de tipos dependentes de máquina, como *port_t*, *segm_t* e *reg_t* (linhas 4525 a 4527) utilizados em processadores Intel, para, respectivamente, endereçar portas de E/S, segmentos de memória e registradores de CPU.

As estruturas, também, podem ser dependentes de máquina. Nas linhas 4537 a 4558, a estrutura *stackframe_s*, que define como os registradores de máquina são salvos na pilha, é definida para processadores Intel. Essa estrutura é extremamente importante — ela é utilizada para salvar e para restaurar o estado interno da CPU sempre que um processo é colocado ou removido do estado "em execução" da Figura 2-2. Definindo-o de uma forma que possa efetivamente ser lida ou escrita por código em linguagem *assembly*, reduz-se o tempo necessário para uma comutação de contexto. *Segdesc_s* é outra estrutura relacionada com a arquitetura dos processadores Intel. É parte do mecanismo de proteção que impede que os processos acessem regiões de memória fora daquelas atribuídas a eles.

Para ilustrar diferenças entre plataformas, algumas definições para a família de processadores Motorola 68000 foram mantidas nesse arquivo. A família de processadores Intel inclui alguns modelos com registradores de 16 bits e outros com registradores de 32 bits, portanto, o tipo *reg_t* básico é *unsigned* para a arquitetura Intel. Para processadores Motorola, *reg_t* é definido como o tipo *u32_t*. Esses processadores também precisam de uma estrutura *stackframe_s* (linhas 4583 a 4603), mas o leiaute é diferente, para tornar as operações de código *assembly* que a utilizam mais rápidas. A arquitetura Motorola não tem nenhuma necessidade dos tipos *port_t* e *segm_t*, ou da estrutura *segdesc_s*. Também há várias estruturas definidas para a arquitetura Motorola que não têm nenhum correspondente Intel.

O próximo arquivo, *proto.h* (linha 4700), é o mais longo arquivo de cabeçalho que veremos. Os protótipos de todas as funções que devem ser conhecidas fora do arquivo em que são definidas estão nesse arquivo. Todos são escritos utilizando a macro *_PROTOTYPE* discutida na seção anterior e, assim, o *kernel* do MINIX pode ser compilado

com um compilador C clássico (Kernighan e Ritchie), como o compilador C original do MINIX, ou com um moderno compilador ANSI Standard C, como o que é parte da versão 2 do MINIX. Alguns desses protótipos são dependentes de sistema, incluindo manipuladores de interrupções e de exceções, e de funções escritas em linguagem *assembly*. Os protótipos de funções necessárias para *drivers* não-discutidos neste texto não são mostrados. O código condicional para processadores Motorola também foi excluído deste e dos demais arquivos que discutiremos.

O último dos cabeçalhos do *kernel* incluídos no cabeçalho mestre é *glo.h* (linha 5000). Encontraremos aqui as variáveis globais do *kernel*. O propósito da macro *EXTERN* foi descrito na discussão sobre *include/minix/const.h*. Normalmente ela se expande em *extern*. Note que muitas definições em *glo.h* são precedidas por essa macro. *EXTERN* é forçado a ser não-definido quando esse arquivo é incluído em *table.c*, onde a macro *_TABLE* é definida. Incluindo *glo.h* em outros arquivos-fonte de C, tornamos as variáveis em *table.c* conhecidas para os outros módulos no *kernel*. *Held_head* e *held_tail* (linhas 5013 e 5014) são ponteiros para uma fila de interrupções pendentes. *Proc_ptr* (linha 5018) aponta para a entrada na tabela de processos para o processo atual. Quando uma chamada de sistema ou uma interrupção ocorre, ele informa onde armazenar os registradores e o estado do processador. *Sig_procs* (linha 5021) informa o número de processos que têm sinais pendentes que ainda não foram enviados ao gerenciador de memória para processamento. Alguns itens em *glo.h* são definidos com *extern* em vez de *EXTERN*. Esses incluem *sizes*, uma matriz preenchida pelo monitor de inicialização, a tabela de tarefas, *tasktab*, e a pilha de tarefas, *t_stack*. Os dois últimos são **variáveis inicializadas**, um recurso da linguagem C. A utilização da macro *EXTERN* não é compatível com inicialização no estilo C, já que uma variável só pode ser inicializada uma vez.

Cada tarefa tem sua própria pilha dentro de *t_stack*. Durante a manipulação de interrupções, o *kernel* utiliza uma pilha separada, mas ela não é declarada aqui, uma vez que só é acessada pela rotina no nível da linguagem *assembly* que gerencia o processamento de interrupções e não precisa ser conhecida globalmente.

Há mais dois arquivos de cabeçalho do *kernel* que são amplamente utilizados, embora não tanto para serem incluídos em *kernel*. O primeiro deles é *proc.h* (linha 5100), que define uma entrada de tabela de processos como a estrutura *proc* (linhas 5110 a 5148). Mais adiante, no mesmo arquivo, ele define a própria tabela de processos como uma matriz dessas estruturas, *proc[NR_TASKS + NR_PROCS]* (linha 5186). Na linguagem C, essa reutilização de um nome é permitida. A macro *NR_TASKS* é definida em *include/minix/const.h* (linha 2953) e *NR PROCS* é definida em *include/minix/config.h* (linha 2639). Juntas, tais macros configuram o tamanho da tabela de processos. *NR_PROCS* poder ser alterada para criar um sistema capaz de gerenciar um número maior de usuários. Como a tabela de processos é acessada freqüentemente, e calcular um endereço em uma matriz requer lentas operações de multiplicação, uma matriz de ponteiros para os elementos da tabela de processos, *pproc_addr* (linha 5187), é utilizada para permitir acesso rápido.

Cada entrada da tabela contém o espaço de armazenamento para os registradores, para o ponteiro de pilha, para o estado, para o mapa de memória, para o limite da pilha, para o id, para a contabilidade, para o tempo do alarme e para informações de mensagens do processo. A primeira parte de cada entrada da tabela de processo é uma estrutura *stackframe_s*. Um processo é colocado em execução carregando em seu ponteiro de pilha o endereço da sua entrada na tabela de processos e lendo todos os registradores da CPU a partir dessa estrutura. Quando um processo não pode completar um SEND porque o destino não está esperando, o remetente é colocado em uma fila apontada pelo campo *p_callerq* do destino (linha 5137). Dessa maneira, quando o destino por fim faz um RECEIVE, é fácil localizar todos os processos que querem enviar para ele. O campo *p_sendlink* (linha 5138) é utilizado para agrupar os membros da fila.

Quando um processo faz um RECEIVE e não há nenhuma mensagem esperando por ele, ele bloqueia, e o número do processo de quem ele quer receber é armazenado em *p_getfrom*. O endereço do buffer de mensagem é armazenado em *p_messbuf*. Os últimos três campos em cada entrada da tabela de processo são *p_nextready*, *p_pending* e *p_pendcount* (linhas 5143 a 5145). O primeiro desses é utilizado para agrupar os processos nas filas do agendador, enquanto o segundo é um mapa de bits utilizado para monitorar sinais que ainda não foram passados para o gerenciador de memória (porque o gerenciador de memória não está esperando uma mensagem). O último campo é uma contabilização desses sinais.

Os bits de sinalizador em *p_flags* definem o estado de cada entrada da tabela. Se qualquer um dos bits está ligado, o processo não pode ser executado. Os vários sinalizadores são definidos e descritos nas linhas 5154 a 5160. Se a entrada não está em uso, *P_SLOT_FREE* é ligado. Após um FORK, *NO_MAP* é ligado para evitar que o processo-filho execute até que seu mapa de memória tenha sido definido. *INSENDING* e *RECEIVING* indicam que o processo está bloqueado tentando enviar ou receber uma mensagem. *PENDING* e *SIG_PENDING* indicam que sinais foram recebidos e *P_STOP* fornece suporte para monitoramento, durante a depuração.

A macro *proc_addr* (linha 5179) é fornecida porque não é possível ter subscritos negativos em C. Logicamente, a matriz *proc* deveria ir de *–NR_TASKS* a *+NR_PROCS*. Infelizmente, em C, ela deve iniciar em 0, portanto, *proc[0]* refere-se à tarefa mais negativa e assim por diante. Para tornar mais fácil monitorar qual entrada acompanha qual processo, podemos escrever

```
rp = proc_addr(n);
```

para atribuir a *rp* o endereço da entrada de processo para o processo n, seja ele positivo ou negativo.

Bill_ptr (linha 5191) aponta para o processo que está sendo cobrado pela CPU. Quando um processo de usuário chama o sistema de arquivos e o sistema de arquivos está executando, *proc_ptr* (em *glo.h*) aponta para o processo do sistema de arquivos. Entretanto, *bill_ptr* apontará para o usuário que está fazendo a chamada, já que o tempo de CPU utilizado pelo sistema de arquivos é contabilizado como tempo de sistema para o processo que fez a chamada.

As duas matrizes *rdy_head* e *rdy_tail* são utilizadas para manter as filas de agendamento. O primeiro processo na fila de tarefas, por exemplo, é apontado por *rdy_head*[*TASK_Q*].

Outro cabeçalho que é incluído em diversos arquivos-fonte é *protect.h* (linha 5200). Quase tudo nesse arquivo trata de detalhes de arquitetura dos processadores Intel que suportam o modo protegido (os 80286, 80386, 80486, Pentium e Pentium Pro). Uma descrição detalhada desses chips está além do escopo deste livro. É suficiente dizer que eles contêm registradores internos que apontam para **tabelas descritoras** na memória. As tabelas descritoras definem como os recursos de sistema são utilizados e evitam que os processos acessem a memória atribuída a outro processo. Além disso, a arquitetura do processador oferece quatro **níveis de privilégio,** dos quais o MINIX tira proveito de três. Esses são definidos simbolicamente nas linhas 5243 a 5245. As partes mais centrais do *kernel*, as partes que executam durante interrupções e que alternam processos, executam com *INTR_PRIVILEGE*. Não há nenhuma parte da memória ou registrador da CPU que não possa ser acessado por um processo com esse nível de privilégio. As tarefas executam no nível *TASK_PRIVILEGE*, que as permite acessar E/S, mas não utilizar instruções que modificam registradores especiais, como aqueles que apontam para tabelas descritoras. Os processos de servidor e de usuário executam no nível *USER_PRIVILEGE*. Os processos que executam nesse nível são incapazes de executar certas instruções, por exemplo aquelas que acessam portas de E/S, alteram atribuições de memória ou alteram os próprios níveis de privilégio. O conceito de níveis de privilégio será familiar para aqueles que conhecem a arquitetura das CPUs modernas, mas aqueles que aprenderam arquitetura de computadores estudando a linguagem *assembly* de microprocessadores baratos podem não ter encontrado essas restrições.

Há vários outros arquivos de cabeçalho no diretório do *kernel*, mas mencionaremos aqui só mais dois. Primeiro, há *sconst.h* (linha 5400), que contém constantes utilizadas pelo código *assembler*. Todas essas são deslocamentos (*offsets*) na parte *stackframe_s* da estrutura de uma entrada na tabela de processo, expressa de uma forma utilizável pelo *assembler*. Como o código *assembler* não é processado pelo compilador C, é mais simples ter essas definições em um arquivo separado. Além disso, como todas essas definições são dependentes de máquina, isolá-las aqui simplifica o processo de portar o MINIX para outro processador que precisará de uma versão diferente de *sconst.h*. Note que muitos deslocamentos são expressos como o valor anterior mais *W*, que é definido como igual ao tamanho de palavra na linha 5401. Isso permite que o mesmo arquivo sirva para compilar uma versão de 16 ou de 32 bits do MINIX.

Há um problema em potencial aqui. Os arquivos de cabeçalho destinam-se a permitir que se forneça um único conjunto correto de definições e então utilizá-las em muitos lugares sem prestar mais atenção aos detalhes. Obviamente, definições duplicadas, como aquelas em *sconst.h*, violam esse princípio. Isso é um caso especial, naturalmente, mas, como tal, atenção especial é requerida se forem feitas mudanças nesse arquivo ou em *proc.h*, para assegurar que os dois arquivos sejam consistentes.

O cabeçalho final que mencionaremos aqui é *assert.h* (linha 5500). O padrão POSIX requer a disponibilidade de uma função *assert*, que pode ser utilizada para fazer um teste em tempo de execução e abortar um programa, imprimindo uma mensagem. De fato, o POSIX requer que um cabeçalho *assert.h* seja fornecido no diretório *include/*, e um é fornecido lá. Então, por que há outra versão aqui? A resposta é que quando algo dá errado em um processo de usuário, pode-se contar com o sistema operacional para fornecer serviços como imprimir uma mensagem no console. Mas se algo dá errado no próprio *kernel*, não se pode contar com os recursos normais do sistema. O *kernel*, assim, oferece suas próprias rotinas para gerenciar *assert* e imprimir mensagens, independentemente das versões na biblioteca normal do sistema.

Há alguns arquivos de cabeçalho em *kernel/* que ainda não discutimos. Eles suportam as tarefas de E/S e serão descritos no próximo capítulo, onde são relevantes. Antes de passar para o código executável, entretanto, vejamos *table.c* (linha 5600), cujo arquivo objeto compilado contém todas as estruturas de dados do *kernel*. Já vimos muitas dessas estruturas de dados definidas, em *glo.h* e *proc.h*. Na linha 5625, a macro *_TABLE* é definida, imediatamente antes das declarações #include. Como é explicado, essa definição faz com que *EXTERN* torne-se definida como uma *string* nula e o espaço de armazenamento seja atribuído a todas as declarações de dados precedidas por *EXTERN*. Além das estruturas em *glo.h* e *proc.h*, o armazenamento para algumas variáveis globais utilizadas pela tarefa de terminal, definidas em *tty.h*, também estão alocadas aqui.

Além das variáveis declaradas em arquivos de cabeçalho, há dois outros lugares onde o armazenamento global de dados é alocado. Algumas definições são feitas diretamente em *table.c*. Nas linhas 5639 a 5674, o espaço de pilha é alocado para cada tarefa. Para cada tarefa opcional, a macro *ENABLE_XXX* correspondente (definida no arquivo *include/minix/config.h)* é utilizada para calcular o tamanho da pilha. Portanto, nenhum espaço é alocado para uma tarefa que não esteja ativa. Seguindo isso, as várias macros *ENABLE_XXX* são utilizadas para determinar se cada tarefa opcional será representada na matriz *tasktab*, composta pelas estruturas *tasktab*, como declaradas anteriormente em *src/kernel/type.h* (linhas 5699 a 5731). Há

um elemento para cada processo que é iniciado durante a inicialização de sistema, seja ele um processo de tarefa, um processo de servidor ou um processo de usuário (i. e., *init*). O índice da matriz implicitamente mapeia entre os números de tarefa e os procedimentos de inicialização associados. *Tasktab* também especifica o espaço de pilha necessário para cada processo e fornece uma *string* de identificação para cada processo. Ele foi colocado aqui em vez de em um arquivo de cabeçalho porque o truque com *EXTERN*, utilizado para impedir múltiplas declarações, não funciona com variáveis inicializadas; isto é, você não pode dizer

```
extern int x = 3;
```

em qualquer lugar. As definições anteriores de tamanho de pilha também permitem alocação de espaço de pilha para todas as tarefas na linha 5734.

Apesar de tentar isolar todas as informações de configuração definíveis pelo usuário em *include/minix/config.h*, é possível ocorrer um erro ao fazer-se coincidir o tamanho da matriz *tasktab* com *NR_TASKS*. No fim de *table.c*, um teste é feito para esse erro, utilizando um pequeno truque. A matriz *dummy_tasktab* é declarada aqui de tal maneira que seu tamanho será impossível e desencadeará um erro de compilador se um engano for feito. Uma vez que a matriz [*dummy*] é declarada como *extern*, nenhum espaço é alocado para ela aqui (ou em qualquer lugar). Como ela não é referenciada em qualquer outro lugar no código, isso não perturbará o compilador.

O outro lugar onde o armazenamento global é alocado está no fim do arquivo de linguagem *assembly mpx386.s* (linha 6483). Essa alocação, sob o rótulo *_sizes*, coloca um número mágico (para identificar um *kernel* MINIX válido) bem no começo do segmento de dados do *kernel*. Espaço adicional está alocado aqui pela pseudo-instrução *.space*. A reserva de espaço de armazenamento dessa maneira pelo programa de linguagem *assembly* torna possível forçar a matriz *_sizes* a localizar-se fisicamente no começo do segmento de dados do *kernel*, tornando fácil programar *boot* para colocar os dados no lugar correto. O monitor de inicialização lê o número mágico e, se estiver correto, sobrescreve-o para iniciar a matriz *_sizes* com os tamanhos das diferentes partes do sistema MINIX. O *kernel* utiliza esses dados durante a inicialização. No momento da inicialização, no que diz respeito ao *kernel*, essa é uma área de dados inicializada. Entretanto, os dados que o *kernel* acaba encontrando não estão disponíveis no momento da compilação. Ele são atualizados pelo monitor de inicialização logo antes de o *kernel* ser inicializado. Isso é algo incomum, normalmente não é necessário escrever programas que conhecem a estrutura interna de outro programa. Mas o período de tempo depois que a energia é aplicada, mas antes de o sistema operacional começar a executar, é incomum e requer técnicas incomuns.

2.6.5 Fazendo a Inicialização do MINIX

Já é quase hora de começar a examinar o código executável. Mas antes de fazer isso, dedicaremos alguns minutos para entender como MINIX é carregado na memória. Ele é, naturalmente, carregado a partir de um disco. A Figura 2-31 mostra como disquetes e discos particionados são organizados.

Quando o sistema é iniciado, o hardware (na realidade, um programa na ROM) lê o primeiro setor do disco de *boot* e executa o código aí localizado. Em um disquete de MINIX não-particionado, o primeiro setor é um bloco de *boot* que carrega o programa de *boot*, como na Figura 2-31(a). Os discos rígidos são particionados, e o programa no primeiro setor lê a tabela de partição, que também está no primeiro setor e carrega e executa o primeiro setor da partição ativa, como mostrado na Figura 2-31(b) (Normalmente uma e apenas uma partição é marcada como ativa). Uma partição do MINIX tem a mesma estrutura que um disquete de MINIX não-particionado, com um bloco de *boot* que carrega o programa de *boot*.

A situação real pode ser um pouco mais complicada do que aquela que a figura mostra porque uma partição pode conter subpartições. Nesse caso, o primeiro setor da partição será outro registro-mestre de inicialização contendo a tabela de partição para as subpartições. Mas, enfim, o controle será passado a um setor de *boot*, o primeiro setor em um dispositivo que não é subdividido ainda mais. Em um disquete, o primeiro setor é sempre um setor de *boot*. O MINIX permite de fato uma forma de particionar um disquete, mas apenas a primeira partição pode ser inicializada; não há registro-mestre de inicialização separado e as subpartições não são possíveis. Isso permite que disquetes particionados e não-particionados sejam montados exatamente do mesmo modo. A principal utilização para um disquete particionado é que ele fornece uma maneira conveniente de dividir um disco de instalação em uma imagem da raiz a ser copiada para um disco de RAM e uma porção montada que pode ser desmontada quando não for mais necessária, a fim de liberar a unidade de disquete para continuar o processo de instalação.

O setor de *boot* do MINIX é modificado no momento em que ele é gravado no disco atualizando os números de setor necessários para localizar um programa chamado *boot* em sua partição ou em sua subpartição. Essa atualização é necessária porque antes do carregamento do sistema operacional não há como utilizar os nomes de diretório e de arquivo para localizar um arquivo. Um programa especial chamado *installboot* é utilizado para fazer a atualização e a gravação do setor de *boot*. *Boot* é o carregador secundário do MINIX. Mas ele pode fazer mais que apenas carregar o sistema operacional, uma vez que ele é um **programa monitor** que permite ao usuário mudar, configurar e sal-

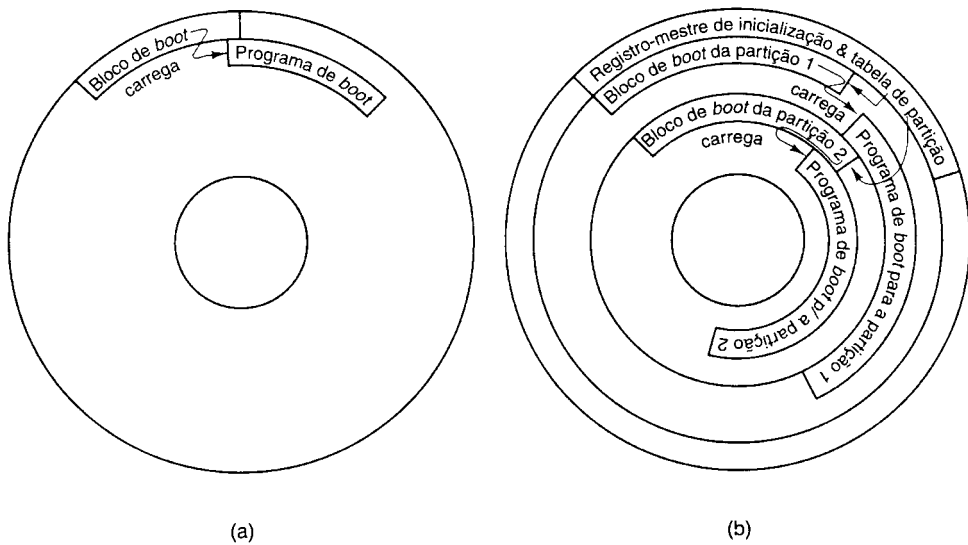



Figura 2-31 As estruturas de disco utilizadas para fazer a inicialização. (a) Disco não-particionado. O primeiro setor é o bloco de *boot*. (b) Disco particionado. O primeiro setor é o registro-mestre de inicialização (*master boot record*).

var vários parâmetros. *Boot* examina o segundo setor de sua partição a fim de localizar um conjunto de parâmetros para utilizar. O MINIX, como o UNIX padrão, reserva o primeiro bloco de 1K de cada dispositivo de disco como um **bloco de *boot*,** mas só um setor de 512 bytes é carregado pelo carregador de *boot* de ROM ou pelo setor de *boot* mestre, portanto, 512 bytes estão disponíveis para salvar as configurações. Essas controlam a operação de *boot* e também são passadas para o próprio sistema operacional. As configurações-padrão apresentam um menu com uma opção, iniciar o MINIX, mas as configurações podem ser modificadas para apresentar um menu mais complexo que permite iniciar outros sistemas operacionais (carregando e executando setores de *boot* de outras partições), ou iniciar o MINIX com várias opções. As configurações-padrão também podem ser modificadas para saltar o menu e para iniciar o MINIX imediatamente.

O *boot* não é uma parte do sistema operacional, mas é suficientemente esperto para utilizar as estruturas de dados do sistema de arquivos para localizar a imagem real do sistema operacional. Por padrão, o *boot* procura um arquivo chamado */minix*, ou, se houver um diretório */minix/*, o arquivo mais recente dentro dele, mas os parâmetros de *boot* podem ser mudados para procurar um arquivo com qualquer nome. Esse grau de flexibilidade não é usual, e a maioria dos sistemas operacionais tem um nome de arquivo predefinido para a imagem do sistema. Mas, o MINIX é um sistema operacional incomum que encoraja os usuários a modificá-lo e a criar novas versões experimentais. A prudência exige que os usuários que fazem isso devem dispor de uma maneira de selecionar múltiplas versões, para serem capazes de retornar para a última versão que funcionou corretamente quando uma experiência fracassa.

A imagem do MINIX carregado por *boot* é nada mais que uma concatenação dos arquivos individuais produzidos pelo compilador quando os programas *kernel*, gerenciador de memória, sistema de arquivos e *init* são compilados. Cada um desses inclui um cabeçalho curto do tipo definido em *include/a.out.h*, e a partir das informações no cabeçalho de cada parte, *boot* determina o espaço a reservar para os dados não-inicializados depois de carregar o código executável e os dados inicializados para cada parte, de modo que a próxima parte possa ser carregada no endereço adequado. A matriz *_sizes*, mencionada na seção anterior, também recebe uma cópia dessas informações para que o *kernel* possa ter acesso às localizações e aos tamanhos de todos os módulos carregados por *boot*. As regiões de memória disponíveis para carregar o setor de *boot*, o próprio *boot* e o MINIX dependerão do hardware. Além disso, algumas arquiteturas de máquina podem requerer adaptação de endereços internos dentro do código executável para corrigi-los para o endereço real onde um programa é carregado. A arquitetura segmentada dos processadores Intel torna isso desnecessário. Como os detalhes do processo de carregamento diferem com o tipo de máquina, e *boot* em si não é parte do sistema operacional, não nos aprofundaremos mais nele aqui. O importante é que de uma maneira ou de outra o sistema operacional é carregado na memória. Uma vez que a carga esteja completa, o controle passa para o código executável do *kernel*.

A propósito, devemos mencionar que sistemas operacionais não são universalmente carregados de discos locais. **Estações de trabalho sem disco** (*diskless workstations*)

podem carregar seus sistemas operacionais a partir de um disco remoto, por uma conexão de rede. Isso requer software de rede em ROM, naturalmente. Embora os detalhes variem em relação ao que descrevemos aqui, de maneira geral, os elementos do processo são semelhantes. O código de ROM deve ser suficientemente inteligente para obter um arquivo executável pela rede que pode, então, obter o sistema operacional completo. Se o MINIX for carregado dessa maneira, muito pouco precisaria ser alterado no processo de inicialização que ocorre uma vez que o código do sistema operacional tenha sido carregado na memória. Naturalmente, ele precisará de um servidor de rede e de um sistema de arquivos modificado que possa acessar arquivos via rede.

2.6.6 Inicialização do Sistema

O MINIX para máquinas tipo IBM PC pode ser compilado no modo de 16 bits se for exigida compatibilidade com chips de processador mais antigos, ou no modo de 32 bits para melhor desempenho em processadores 80386 ou superiores. O mesmo código-fonte em C é utilizado e o compilador gera a saída apropriada, que depende de o compilador ser da versão de 16 ou de 32 bits. Uma macro definida pelo compilador determina a definição da macro *_WORD_ SIZE* em *include/minix/config.h*. A primeira parte do MINIX a executar é escrita em linguagem *assembly*, e diferentes arquivos de código-fonte devem ser utilizados para os compiladores de 16 ou de 32 bits. A versão de 32 bits do código de inicialização está em *mpx386.s*. A alternativa, para sistemas de 16 bits, está em *mpx88.s*. Ambos também incluem suporte de linguagem *assembly* para outras operações de baixo nível do *kernel*. A seleção é feita automaticamente em *mpx.s*. Esse arquivo é tão curto que o arquivo inteiro pode ser apresentado na Figura 2-32.

Mpx.s mostra uma utilização incomum da declaração #include do pré-processador C. Habitualmente #include é utilizado para incluir arquivos de cabeçalho, mas também pode ser utilizado para selecionar uma seção alternativa do código-fonte. Utilizar declarações #if para fazer isso exigiria colocar todo o código dos dois grandes arquivos *mpx88.s* e *mpx386.s* em um único arquivo. Não apenas isso seria de difícil manejo como também desperdiçaria espaço em disco, uma vez que, em uma instalação particular, é possível que um desses dois arquivos absolutamente não seja utilizado e possa ser arquivado ou excluído. Na discussão a seguir utilizaremos o *mpx386.s*, 32 bits, como exemplo.

Como esse é nosso primeiro exame do código executável, vamos começar com algumas palavras sobre como faremos esse exame ao longo de todo este livro. O grande número de arquivos-fonte utilizados na compilação de um programa em C de tamanho razoável pode ser difícil de acompanhar. Em geral, manteremos as discussões restritas a um único arquivo por vez e seguiremos em ordem pelos arquivos. Iniciaremos com o ponto de entrada para cada parte do sistema do MINIX e seguiremos a linha de execução principal. Quando uma chamada para uma função de suporte for encontrada, diremos algumas palavras sobre o propósito da chamada, mas normalmente não entraremos em uma descrição detalhada dos aspectos internos da função nesse ponto, deixando isso para quando chegarmos à definição da função chamada. Funções subordinadas importantes normalmente são definidas no mesmo arquivo em que são chamadas, seguindo as funções de chamada de nível mais alto, mas as pequenas funções de propósito geral, às vezes, são reunidas em arquivos separados. Além disso, tentamos ao máximo possível organizar o código dependente de máquina em arquivos separados do código independente de máquina para facilitar a portabilidade para outras plataformas. Um esforço significativo foi dedicado para organizar o código, e, de fato, muitos arquivos foram reescritos no curso da redação deste texto para organizá-los melhor para o leitor. Mas um programa grande tem muitas ramificações e, às vezes, entender uma função principal requer a leitura das funções que ela chama, portanto, ter algumas tiras de papel à mão para usarmos como marcadores e desviar de nossa ordem de discussão para ver as coisas em uma ordem diferente, às vezes, pode ser útil.

Tendo exposto a maneira como organizamos a discussão sobre o código, devemos iniciar imediatamente justificando uma exceção importante. A inicialização do MINIX envolve várias transferências de controle entre as rotinas de linguagem *assembly* em *mpx386.s* e rotinas escritas em C localizadas nos arquivos *start.c* e *main.c*. Descreveremos essas rotinas na ordem em que elas são executadas, mesmo que isso envolva pular de um arquivo para outro.

Uma vez que o processo de inicialização carregou o sistema operacional na memória, o controle é transferido para o rótulo MINIX (em *mpx386.s*, linha 6051). A primeira instrução é um salto sobre alguns bytes de dados; isso inclui

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figura 2-32 Como os arquivos-fonte alternativos de linguagem *assembly* são selecionados.

os sinalizadores do monitor de *boot* (linha 6054), utilizados pelo monitor de *boot* para identificar várias características do *kernel*, sobretudo se é um sistema de 16 ou de 32 bits. O monitor de *boot* sempre inicia no modo de 16 bits, mas alterna a CPU para o modo de 32 bits se necessário. Isso acontece antes da passagem do controle para o MINIX. O monitor também configura uma pilha. Há uma quantidade substancial de trabalho a ser feito pelo código de linguagem *assembly*: configurar uma estrutura de pilha para oferecer o ambiente adequado para código gerado pelo compilador de C, copiar as tabelas utilizadas pelo processador para definir segmentos de memória e configurar vários registradores do processador. Logo que esse trabalho completa-se, o processo de inicialização continua por meio da chamada (na linha 6109) da função em C *cstart*. Note que essa é referida como *_cstart* no código de linguagem *assembly*. Isso porque todas as funções compiladas pelo compilador C tem um sublinhado precedendo seus nomes nas tabelas de símbolos, e o *linkeditor* procura por esses nomes quando módulos compilados separadamente são vinculados. Como o *assembler* não adiciona sublinhados, o escritor de um programa em linguagem *assembly* deve explicitamente adicionar um para o *linkeditor* ser capaz de localizar o nome correspondente no arquivo-objeto gerado pelo compilador de C. *Cstart* chama outra rotina para iniciar na **Tabela Global de Descritores** a estrutura central de dados utilizada por processadores Intel de 32 bits para supervisionar a proteção de memória e a **Tabela de Descritores de Interrupções**, utilizada para selecionar o código a ser executado para cada possível tipo de interrupção. Ao retornar de *cstart*, as instruções lgdt e lidt (linhas 6115 e 6116) tornam essas tabelas efetivas carregando os registradores dedicados por meio dos quais elas são endereçadas. A seguinte instrução,

```
jmpf CS_SELECTOR:csinit
```

parece, à primeira vista, uma não-operação, uma vez que transfere o controle para exatamente onde o controle estaria se houvesse uma série de instruções *nop* em seu lugar. Mas isso é uma parte importante do processo de inicialização. Esse salto força a utilização das estruturas que acabaram de ser iniciadas. Após mais alguma manipulação dos registradores do processador, o MINIX termina com um salto (não uma chamada) da linha 6131 para o ponto de entrada principal do *kernel* (em *main.c*). Nesse ponto, o código de inicialização em *mpx386.s* está concluído. O resto do arquivo contém o código para iniciar ou para reiniciar uma tarefa ou processo, manipuladores de interrupção e outras rotinas de suporte que tiveram de ser escritas em linguagem *assembly* por razões de eficiência. Retornaremos a elas na próxima seção.

Agora veremos as funções de inicialização de alto nível em C. A estratégia geral é utilizar tanto quanto possível código de alto nível em C. Já há duas versões do código *mpx*, como vimos, e qualquer coisa que possa ser transportada para o código em C elimina dois blocos de código *assembler*. A primeira coisa feita por *cstart* (em *start.c*, linha 6524) é configurar os mecanismos de proteção da CPU e as tabelas de interrupção, chamando *prot_init*. Então, ela faz coisas como copiar os parâmetros de *boot* para a parte de memória do *kernel* e convertê-los em valores numéricos. Ela também determina o tipo de monitor de vídeo, o tamanho da memória, o tipo de máquina, o modo de operação do processador (real ou protegido) e se um retorno para o monitor de *boot* é possível. Todas as informações são armazenadas em variáveis globais apropriadas, para acesso quando necessário por qualquer parte do código do *kernel*.

Main (em *main.c*, linha 6721) completa a inicialização e, então, inicia a execução normal do sistema. Ela configura o hardware de controle de interrupção chamando *intr_init*. Isso é feito aqui porque não pode sê-lo até que o tipo de máquina seja conhecido, e o procedimento está em um arquivo separado porque é muito dependente do hardware. O parâmetro (1) na chamada informa *intr_init* de que ele está inicializando para o MINIX. Com um parâmetro (0), ele pode ser chamado para reinicializar o hardware para o estado original. A chamada a *intr_init* também dá dois passos para assegurar que qualquer interrupção que ocorra antes que a inicialização esteja completa não tenha nenhum efeito. Primeiro um byte é gravado para cada chip controlador de interrupção para inibir a resposta para entrada externa. Então, todas as entradas na tabela utilizadas para acessar os manipuladores de interrupção específicos de cada dispositivo são preenchidas com o endereço de uma rotina que inofensivamente imprimirá uma mensagem se uma interrupção espúria for recebida. Mais tarde, essas entradas da tabela serão substituídas, uma a uma, por ponteiros para as rotinas dos manipuladores, uma vez que cada uma das tarefas de E/S executa sua própria rotina de inicialização. Cada tarefa então redefinirá um bit no chip controlador de interrupção para ativar sua própria entrada de interrupção.

Mem_init é chamada em seguida. Ela inicializa uma matriz que define a localização e o tamanho de cada bloco de memória disponível no sistema. Como acontece na inicialização do hardware de interrupção, os detalhes dependem do hardware, e o isolamento de *mem_init* como uma função em um arquivo separado mantém *main* livre de código que não é portável para hardware diferente.

A parte maior do código de *main* é dedicada a configurar a tabela de processos para que quando as primeiras tarefas e processos forem agendados, seus mapas de memória e seus registradores estejam configurados corretamente. Todas as entradas na tabela de processos são marcadas como livres, e a matriz *pproc _addr* que acelera o acesso à tabela de processos é inicializada pelo laço nas linhas 6745 a 6749. O código na linha 6748,

```
(pproc_addr+ NR_TASKS)[t] = rp;
```

poderia igualmente ter sido definido como

```
pproc_addr[t + NR_TASKS] = rp;
```

porque na linguagem C *a*[*i*] é somente outro meio de escrever *(*a* +*i*). Então, não faz muita diferença se você adicionar uma constante para *a* ou para *i*. Alguns compiladores de C geram um código ligeiramente melhor se você adicionar uma constante a matriz em vez de ao índice.

A parte maior de *main*, o longo laço das linhas 6762 a 6815, inicializa a tabela de processos com as informações necessárias para executar as tarefas, os servidores e o *init*. Todos esses processos devem estar presentes no momento da inicialização e nenhum deles terminará durante a operação normal. No início do laço, o endereço de uma entrada de tabela de processos é atribuído a *rp* (linha 6763). Como *rp* é um ponteiro para uma estrutura, os elementos da estrutura podem ser acessados utilizando uma notação como *rp–>p_name*, como é feito na linha 6765. Essa notação é utilizada extensamente no código-fonte do MINIX.

As tarefas, naturalmente, são todas compiladas no mesmo arquivo que o *kernel*, e as informações sobre seus requisitos de pilha estão na matriz *tasktab* definida em *table.c*. Como as tarefas são compiladas no *kernel* e podem chamar código e acessar dados localizados em qualquer lugar no espaço do *kernel*, o tamanho de uma tarefa individual não é significativo, e o campo de tamanho de cada uma delas é preenchido com os tamanhos do próprio *kernel*. A matriz *sizes* contém os tamanhos de texto e de dados em *cliks* do *kernel*, gerenciador de memória, sistema de arquivos e *init*. Essas informações são atualizadas na área de dados do *kernel* por *boot* antes de o *kernel* começar a executar e aparecem para o *kernel* como se o compilador as tivesse fornecido. Os primeiros dois elementos de *sizes* são os tamanhos de texto e de dados do *kernel*; os dois seguintes são o gerenciador de memória e assim por diante. Se qualquer um dos quatro programas não utiliza espaço I e D separados, o tamanho do texto é 0; e texto e dados são agrupados juntos como dados. Atribuir a *sizeindex* um valor de zero (linha 6775) para cada uma das tarefas garante que o elemento zero de *sizes* nas linhas 6783 e 6784 será acessado para todas as tarefas. A atribuição a *sizeindex* na linha 6778 dá a cada um dos servidores e a *init* seu próprio índice em *sizes*.

O projeto original do IBM PC colocava a memória ROM no topo do intervalo de memória utilizável, que é limitado a 1MB em uma CPU 8088. Máquinas modernas compatíveis com PC sempre têm mais memória do que o PC original, mas, por razões de compatibilidade, elas ainda têm memória ROM nos mesmos endereços que as máquinas mais antigas. Assim, a memória de leitura-escrita é descontínua, com um bloco de ROM entre os 640KB inferiores e o intervalo superior acima de 1MB. O monitor de *boot* carrega os servidores e *init* no intervalo de memória acima da ROM se possível. Isso é feito principalmente para o benefício do sistema de arquivos, assim um *cache* de bloco maior pode ser utilizado sem colidir com a memória ROM. O código condicional nas linhas 6804 a 6810 assegura que essa utilização da área de memória alta é registrada na tabela de processos.

Duas entradas na tabela de processos correspondem a processos que não precisam ser agendados da maneira usual. Esses processos são *IDLE* e *HARDWARE*. *IDLE* é um laço que não faz nada e é executado quando não há mais nada pronto para executar; o processo *HARDWARE* existe para fins de contabilidade — ele é creditado com o tempo utilizado para servir uma interrupção. Todos os demais processos são colocados nas filas apropriadas pelo código na linha 6811. A função chamada, *lock_ready*, configura uma variável de bloqueio, *switching*, antes de modificar as filas e, então, remover o bloqueio quando a fila foi modificada. O bloqueio e o desbloqueio não são requeridos neste ponto, quando nada ainda está executando, mas esse é o método-padrão e não há motivo para criar código extra para ser utilizado apenas uma única vez.

O último passo na inicialização de cada entrada na tabela de processos é chamar *alloc_segments*. Esse procedimento é parte da tarefa de sistema, mas naturalmente nenhuma tarefa ainda está executando, e ele é chamado como um procedimento normal na linha 6814. É uma rotina dependente de máquina que configura nos campos adequados as localizações, os tamanhos e os níveis de permissão para os segmentos de memória utilizados por cada processo. Para processadores Intel mais antigos, que não suportam modo protegido, ele define só as localizações de segmento. Ele teria de ser reescrito para gerenciar um tipo de processador com um método diferente de alocação de memória.

Uma vez que a tabela de processos é inicializada para todas as tarefas, os servidores e *init*, o sistema está praticamente pronto para rodar. A variável *bill_ptr* informa qual processo é cobrado pelo tempo de processador; ela precisa ter um valor inicial configurado na linha 6818, e *IDLE* é uma escolha apropriada. Mais adiante, ele pode ser alterado pela próxima função chamada, *lock_pick_proc*. Todas as tarefas agora estão prontas para executar e *bill_ptr* será alterado quando um processo de usuário executar. Outro trabalho de *lock_pick_proc* é fazer variável *proc_ptr* apontar para a entrada na tabela de processos do próximo processo a ser executado. Essa seleção é feita examinando as filas de tarefas, de servidores e de processos de usuário, nessa ordem. Nesse caso, o resultado é apontar *proc_ptr* para o ponto de entrada da tarefa de console, que é sempre a primeira a ser iniciada.

Por fim, *main* entra no seu curso. Em muitos programas em C, *main* é um laço, mas no *kernel* do UNIX seu trabalho está concluído, uma vez que a inicialização está completa. A chamada a *restart* na linha 6822 inicia a primeira tarefa. O controle nunca retornará a *main*.

_Restart é uma rotina de linguagem *assembly* em *mpx386.s*. De fato, *_restart* não é uma função completa; é um ponto de entrada intermediário em um procedimento maior. Nós a discutiremos em detalhe na próxima seção; por enquanto diremos simplesmente que *_restart* cau-

sa uma comutação de contexto, de modo que o processo apontado por *proc_ptr* executará. Quando *_restart* executar pela primeira vez, poderemos dizer que o MINIX está executando — ele está executando um processo. *_Restart* é executado repetidas vezes à medida que as tarefas, os servidores e os processos de usuário recebem suas oportunidades para executar e, então, são suspensos, seja para esperar entrada, seja para dar vez a outro processo.

A primeira tarefa enfileirada (a que utiliza a entrada 0 da tabela de processos, ou seja, aquela com o número mais negativo) é sempre a tarefa de console, então, outras tarefas podem utilizá-la para informar progresso ou problemas ao iniciarem. Ela executa até que bloqueie tentando receber uma mensagem. Então, a próxima tarefa executará até que ela, também, bloqueie tentando receber uma mensagem. Por fim, todas as tarefas serão bloqueadas, de modo que o gerenciador de memória e o sistema de arquivos podem executar. Ao executar pela primeira vez, estes dois últimos farão alguma inicialização, mas ambos acabarão bloqueando também. Por fim, *init* criará um processo *getty* para cada terminal. Esses processos bloquearão até que uma entrada seja digitada em algum terminal, ponto em que o primeiro usuário pode efetuar *login*.

Acabamos de traçar a inicialização do MINIX por três arquivos, dois escritos em C e um em linguagem *assembly*. O arquivo em linguagem *assembly*, *mpx386.s*, contém código adicional utilizado no tratamento de interrupções, que veremos na próxima seção. Entretanto, antes de prosseguir, vamos encerrar esta parte com uma breve descrição das rotinas restantes nos dois arquivos em C. Os outros procedimentos em *start.c* são *k_atoi* (linha 6594), que converte uma *string* em um número inteiro, e *k_getenv* (linha 6606), que é utilizado para localizar entradas no ambiente do *kernel*, que é uma cópia dos parâmetros de *boot*. Essas são versões simplificadas das funções-padrão de biblioteca que foram reescritas aqui para manter o *kernel* simples. O único procedimento que resta em *main.c* é *panic* (linha 6829). Ele é chamado quando o sistema descobre uma condição que torna impossível continuar. Condições típicas de *panic* são um bloco crítico de disco que se tornou ilegível, um estado interno inconsistente que foi detectado ou uma parte do sistema que chamou outra parte com parâmetros inválidos. As chamadas para *printf* aqui são realmente chamadas para a rotina *printk* do *kernel*, de modo que o *kernel* pode imprimir no console mesmo que a comunicação interprocesso normal seja interrompida.

2.6.7 Tratamento de Interrupções no MINIX

Os detalhes do hardware de interrupção são dependentes de sistema, mas qualquer sistema deve ter elementos funcionalmente equivalentes a esses descritos para sistemas com CPUs Intel de 32 bits. As interrupções geradas por dispositivos de hardware são sinais elétricos e são gerenciados em primeiro lugar por um controlador de interrupções, um circuito integrado que pode capturar diversos desses sinais e gera para cada um deles um padrão único de dados no barramento de dados do processador. Isso é necessário porque o processador em si tem apenas uma entrada para detectar todos esses dispositivos e assim não pode diferenciar qual dispositivo precisa de serviço. Os PCs que utilizam processadores Intel de 32 bits normalmente são equipados com dois desses chips controladores. Cada um pode gerenciar oito entradas, mas um deles é um escravo que alimenta com sua saída uma das entradas do mestre, portanto 15 dispositivos externos distintos podem ser detectados pela combinação, como mostrado na Figura 2-33.

Na figura, os sinais de interrupção chegam às várias linhas *IRQ n* mostradas à direita. A conexão com o pino INT da CPU informa ao processador que uma interrupção ocorreu. O sinal INTA (reconhecimento de interrupção) da CPU faz com que o controlador responsável pela interrupção coloque dados no barramento de dados do sistema informando ao processador qual rotina de serviço executar. O chip controlador de interrupção é programado durante a inicialização do sistema, quando *main* chama *intr_init*. A programação determina a saída enviada para a CPU por um sinal recebido em cada uma das linhas de entrada, assim como vários outros parâmetros de operação do controlador. O dado colocado no barramento é um número de 8 bits, utilizado para indexar em uma tabela com até 256 elementos. A tabela do MINIX tem 56 elementos. Destes, 35 realmente são utilizados; os outros são reservados para utilização em futuros processadores Intel ou para aprimoramentos futuros do MINIX. Em processadores Intel de 32 bits, essa tabela contém descritores o portão de interrupção, cada um dos quais é uma estrutura de 8 bytes com vários campos.

Há vários modos de resposta possíveis a interrupções; no utilizado pelo MINIX, os campos de maior interesse para nós em cada um dos descritores de portão de interrupção apontam para o segmento de código executável da rotina de serviço e para o endereço inicial dentro dele. A CPU executa o código apontado pelo descritor selecionado. O resultado é exatamente o mesmo que executar uma instrução

```
int <nnn>
```

em linguagem *assembly*. A única diferença é que no caso de uma interrupção de hardware o <*nnn*> origina de um registrador no chip controlador de interrupção, e não de uma instrução na memória do programa.

O mecanismo de comutação de tarefas de um processador Intel de 32 bits que é chamado para executar em resposta a uma interrupção é complexo, e alterar o contador de programa para executar outra função é só uma parte dele. Quando a CPU recebe uma interrupção enquanto está executando um processo, ela define uma nova pilha para utilizar durante o serviço de interrupção. A localização dessa pilha é determinada por uma entrada no **Segmento de Estado de Tarefa** (*Task State Segment*, TSS). Há uma estrutura dessas para o sistema inteiro, inicializa-

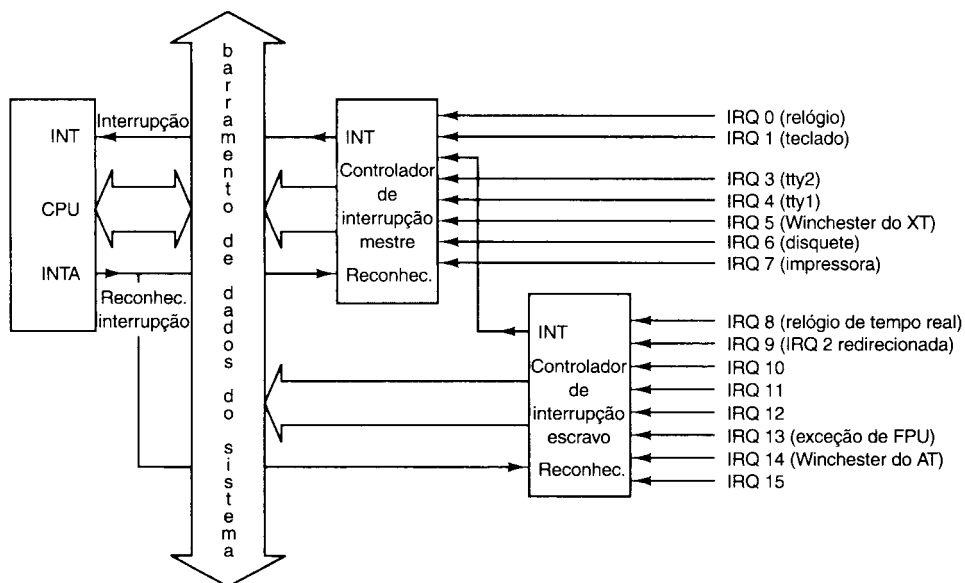



Figura 2-33 Hardware de processamento de interrupções em um PC Intel de 32 bits.

da por uma chamada de *cstart* a *prot_init*, e modificada à medida que cada processo é iniciado. O efeito é que a nova pilha criada por uma interrupção sempre inicia no fim da estrutura *stackframe_s* dentro da entrada da tabela de processo do processo interrompido. A CPU automaticamente coloca vários registradores nessa nova pilha, incluindo aqueles necessários para restaurar a pilha do próprio processos interrompido e restaurar seu contador-programa. Quando o código do manipulador de interrupção começa a executar, ele utiliza essa área na tabela de processos como sua pilha, e muitas das informações necessárias para retornar ao processo interrompido já terão sido armazenadas. O manipulador de interrupção coloca na pilha o conteúdo dos registradores adicionais, preenchendo a estrutura de pilha e, então, alterna para uma pilha fornecida pelo *kernel* enquanto faz o que for necessário para servir a interrupção.

O término de uma rotina de serviço de interrupção é feito alternando a pilha, a partir da pilha do *kernel* de volta a uma estrutura de pilha na tabela de processos (mas não necessariamente a mesma que foi criada pela última interrupção), retirando da pilha explicitamente os registradores adicionais e executando uma instrução iretd (retorno de interrupção). Iretd restaura o estado que existia antes da interrupção, restaurando os registradores que foram colocados na pilha pelo hardware e comuta para a pilha que estava em utilização antes da interrupção. Portanto, uma interrupção pára um processo, e a conclusão do serviço de interrupção reinicia um processo, possivelmente um diferente daquele que foi recentemente interrompido. Diferentemente dos mecanismos de interrupção mais simples, que são o assunto normal dos textos de programação de linguagem *assembly*, nada é armazenado na pilha do processo interrompido durante uma interrupção. Além disso, como a pilha é criada de novo em uma localização conhecida (determinada pelo TSS) depois de uma interrupção, o controle de múltiplos processos é simplificado. Para iniciar um processo diferente, tudo que é necessário é apontar o ponteiro da pilha para a estrutura de pilha de outro processo, retirada da pilha dos registradores que explicitamente foram lá acrescentados, e executar uma instrução iretd.

A CPU desativa todas as interrupções quando recebe uma interrupção. Isso garante que não ocorrerá nada que possa causar o estouro da estrutura de pilha dentro de uma entrada na tabela de processos. Isso é automático, mas também existem instruções no nível *assembly* para desativar e para ativar interrupções. O manipulador de interrupção reativa as interrupções depois de alternar para a pilha do *kernel*, localizada fora da tabela de processos. Ele deve desativar todas as interrupções novamente antes de alternar de volta para uma pilha dentro da tabela de processos, naturalmente, mas enquanto está manipulando uma interrupção, outras interrupções podem ocorrer e serem processadas. A CPU monitora interrupções aninhadas e emprega um método simplificado para alternar de uma rotina de serviço de interrupção e retornar de uma quando um manipulador de interrupção é interrompido. Quando uma nova interrupção é recebida enquanto o manipulador (ou outro código do *kernel*) está executando, uma nova pilha não é criada. Em vez disso, a CPU coloca na pilha existente registradores essenciais necessários para retomar o código

interrompido. Quando um iretd é encontrado durante a execução do código do *kernel*, um mecanismo simplificado de retorno é utilizado também. O processador pode determinar como gerenciar o iretd examinando o seletor de segmento de código que é retirado da pilha como parte da ação de iretd.

Os níveis de privilégio mencionados anteriormente controlam as diferentes respostas a interrupções recebidas enquanto um processo está executando ou enquanto o código do *kernel* (incluindo rotinas de serviço de interrupção) está executando. O mecanismo mais simples é utilizado quando o nível de privilégio do código interrompido é o mesmo que o nível de privilégio do código sendo executado em resposta à interrupção. É só quando o código interrompido tem privilégio inferior ao código do serviço de interrupção que o mecanismo mais elaborado, utilizando o TSS e uma nova pilha, é empregado. O nível de privilégio de um segmento de código é registrado no seletor de segmento de código e como este é um dos itens empilhados durante uma interrupção, ele pode ser examinado ao retornar da interrupção para determinar o que a instrução iretd deve fazer. Outro serviço é fornecido pelo hardware quando uma nova pilha é criada para uso enquanto servindo uma interrupção. O hardware faz uma verificação para assegurar-se de que a nova pilha é suficientemente grande para, pelo menos, suportar a quantidade mínima de informações que deve ser colocada nela. Isso evita que o código mais privilegiado do *kernel* seja acidentalmente (ou maliciosamente) derrubado por um processo de usuário fazendo uma chamada de sistema com uma pilha inadequada. Esses mecanismos são construídos no processador especificamente para uso na implementação de sistemas operacionais que suportam múltiplos processos.

Esse comportamento pode causar confusão se você não estiver familiarizado com o funcionamento interno das CPUs Intel de 32 bits. Normalmente tentamos evitar descrever tais detalhes, mas entender o que acontece quando uma interrupção ocorre e quando uma instrução iretd é executada é essencial para compreender como o *kernel* controla as transições para e a partir do estado "em execução" da Figura 2-2. O fato de que o hardware gerencia grande parte do trabalho torna a vida muito mais fácil para o programador e presumivelmente torna o sistema resultante mais eficiente. Toda essa ajuda do hardware, entretanto, torna realmente muito difícil entender o que está acontecendo a partir da simples leitura do software.

Só uma parte minúscula do *kernel* do MINIX realmente vê as interrupções de hardware. Esse código está em *mpx386.s*. Aqui há um ponto de entrada para cada interrupção. O código-fonte em cada ponto de entrada, *_hwint00* a *_hwint07*, (linhas 6164 a 6193) parece-se com uma chamada a *hwint_master* (linha 6143), e os pontos de entrada *_hwint08* a *_hwint15* (linhas 6222 a 6251) parecem-se com chamadas a *hwint_slave* (linha 6199). Cada ponto de entrada parece passar um parâmetro na chamada, indicando qual dispositivo precisa de serviço. De fato, essas não são realmente chamadas, mas macros, e oito cópias separadas do código definido pela definição de macro de *hwint_master* são montados com apenas o parâmetro *irq* diferente. De maneira semelhante, oito cópias da macro *hwint_slave* são montadas. Isso pode parecer extravagante, mas o código montado é bastante compacto. O código-objeto de cada macro expandida ocupa menos de 40 bytes. Ao servir uma interrupção, a velocidade é importante e fazendo dessa maneira eliminamos o acréscimo de executar código para carregar um parâmetro, chamar uma sub-rotina e recuperar o parâmetro.

Continuaremos a discussão sobre *hwint_master* como se ela realmente fosse uma única função em vez de uma macro que é expandida em oito lugares diferentes. Lembre-se de que antes de *hwint_master* começar a executar, a CPU criou uma nova pilha na estrutura de pilha do processo interrompido, dentro de sua entrada na tabela de processos e que vários registradores-chave já foram salvos aí. A primeira ação de *hwint_master* é chamar *save* (linha 6144). Essa sub-rotina coloca na pilha todos os outros registradores necessários para reiniciar o processo interrompido. *Save* poderia ter sido escrita *inline** como parte da macro para aumentar a velocidade, mas isso teria mais que dobrado o tamanho da macro e, em qualquer caso, *save* é necessária para chamadas por outras funções. Como veremos, *save* faz alguns truques com a pilha. Ao retornar para *hwint_master*, a pilha do *kernel*, não uma estrutura de pilha na tabela de processos, está em utilização. O próximo passo é manipular o controlador de interrupções, para impedi-lo de receber outra interrupção da origem que gerou a atual interrupção (linhas 6145 a 6147). Essa operação mascara a capacidade do chip controlador de responder a uma entrada em particular; a capacidade da CPU de responder a todas interrupções é inibida internamente quando ela recebe pela primeira vez o sinal de interrupção e ainda não foi restaurada nesse ponto.

O código nas linhas 6148 a 6150 redefine o controlador de interrupção e, então, ativa a CPU novamente para receber interrupção de outras fontes. Em seguida, o número da interrupção que está sendo servida é utilizado pela instrução call indireta na linha 6152 como índice em uma tabela de endereços das rotinas de baixo nível específicas de cada dispositivo. Chamamos essas rotinas de baixo nível, mas elas são escritas em C e, em geral, executam operações como servir um dispositivo de entrada e transferir os dados para um buffer onde podem ser acessados quando a tarefa correspondente tem sua próxima chance de executar. Uma quantidade substancial de processamento pode acontecer antes do retorno dessa chamada.

Veremos exemplos de código de *driver* de baixo nível no próximo capítulo. Entretanto, para entender o que está acontecendo aqui em *hwint_master*, mencionamos agora que o código de baixo nível pode chamar *interrupt* (em *proc.c*, que discutiremos na próxima seção) e que *interrupt* transforma a interrupção em uma mensagem para a

*N. de R. Técnica em que cada chamada de rotina é substituída pelo seu código completo.

tarefa que serve o dispositivo que causou a interrupção. Além disso, uma chamada a *interrupt* invoca o agendador e pode selecionar essa tarefa para executar em seguida. Ao retornar da chamada para o código específico ao dispositivo, a capacidade do processador de responder a todas interrupções é novamente desativada, pela instrução cli na linha 6154, e o controlador de interrupção é preparado para ser capaz de responder ao dispositivo que causou a atual interrupção quando todas as interrupções forem reativadas em seguida (linhas 6157 a 6159). Então, *hwint_master* termina com uma instrução ret (linha 6160). Não é óbvio que um truque acontece aqui. Se um processo foi interrompido, a pilha em utilização neste ponto é a pilha do *kernel* e não a pilha dentro da tabela de processos que foi configurada pelo hardware antes de *hwint_master* ser iniciado. Nesse caso, a manipulação da pilha por *save* terá deixado o endereço de *_restart* na pilha do *kernel*. Isso faz com que uma tarefa, um servidor ou um processo de usuário execute mais uma vez. Esse pode não ser, e aliás é improvável que seja, o mesmo processo que estava executando originalmente; isso depende do processamento da mensagem criado pela rotina de serviço de interrupção específica do dispositivo ter causado uma alteração no processo de agendamento. Esse, então, é o coração do mecanismo que cria a ilusão de múltiplos processos executando simultaneamente.

Para concluir, devemos mencionar que quando uma interrupção ocorre enquanto o código do *kernel* está executando, a pilha do *kernel* já está em utilização e *save* deixa o endereço de *restart1* na pilha do *kernel*. Nesse caso, o que quer que o *kernel* estivesse fazendo anteriormente continua após o ret no fim de *hwint_master*. Portanto, as interrupções podem ser aninhadas, mas quando todas as rotinas de serviço de baixo nível estão completas *_restart* por fim executará, e um processo diferente do que foi interrompido pode ser colocado em execução.

Hwint_slave (linha 6199) é muito semelhante a *hwint_master*, exceto que deve reativar ambos os controladores, mestre e escravo, uma vez que os dois foram desativados pela recepção de uma interrupção pelo escravo. Há alguns aspectos sutis da linguagem *assembly* que são vistos aqui. Primeiro, na linha 6206 há uma linha

```
jmp   .+ 2
```

que especifica um salto cujo endereço-alvo é a instrução imediatamente seguinte. Essa instrução está colocada aqui unicamente para adicionar uma pequena pausa. Os autores do BIOS original do IBM PC consideraram uma pausa necessária entre instruções de E/S consecutivas e estamos seguindo seu exemplo, embora essa pausa possa não ser necessária nos atuais computadores compatíveis com IBM PC. Esse tipo de ajuste fino é uma boa razão por que programar dispositivos de hardware seja considerado um ofício esotérico por algumas pessoas. Na linha 6214, há um salto condicional para uma instrução com um rótulo numérico,

```
0:   ret
```

a ser encontrado na linha 6218. Note que a linha

```
jz 0f
```

não especifica um número de bytes a saltar, como no exemplo anterior. O *0f* aqui não é um número hexadecimal. Essa é a maneira como o *assembler* utilizado pelo compilador MINIX especifica um **rótulo local**; o *0f* significa um salto **para frente** do próximo rótulo numerado com 0. Nomes de rótulo normais não podem começar com caracteres numéricos. Outro interessante e possivelmente confuso ponto é que o mesmo rótulo ocorre em outra parte do mesmo arquivo, na linha 6160 em *hwint_master*. A situação é ainda mais complicada do que parece à primeira vista, uma vez que esses rótulos estão dentro de macros e as macros são expandidas antes de o montador ver esse código. Assim, há realmente 16 rótulos 0: no código visto pelo montador. A possível proliferação de rótulos declarados dentro de macros é, de fato, a razão por que a linguagem *assembly* fornece rótulos locais; ao resolver um rótulo local, o montador utiliza aquele mais próximo que coincide com a direção especificada, e as outras ocorrências de um rótulo local são ignoradas.

Agora passemos a examinar *save* (linha 6261), que já mencionamos várias vezes. Seu nome descreve uma de suas funções, que é salvar o contexto do processo interrompido na pilha fornecida pela CPU, que é uma estrutura de pilha dentro da tabela de processos. *Save* utiliza a variável *_k_reenter* para contar e para determinar o nível de aninhamento das interrupções. Se um processo estava executando quando a atual interrupção ocorreu, a instrução

```
mov    esp, k_stktop
```

na linha 6274 alterna para a pilha do *kernel*, e a instrução seguinte coloca na pilha o endereço de *_restart* (linha 6275). Caso contrário, a pilha do *kernel* já está em uso e o endereço de *restart1* é colocado na pilha (linha 6281). Em qualquer caso, com uma pilha em utilização possivelmente diferente da que estava em efeito na entrada, e com o endereço de retorno na rotina que a chamou enterrado embaixo dos registradores que ela acabou de colocar na pilha, uma simples instrução *return* não é adequada para retornar para o procedimento que fez a chamada. As instruções

```
jmp    RETADR-P_STACKBASE(eax)
```

que terminam os dois pontos de saída de *save*, na linha 6277 e na linha 6282, utilizam o endereço que foi colocado na pilha quando *save* foi chamada.

O próximo procedimento em *mpx386.s* é *_s_call*, que começa na linha 6288. Antes de examinar seus detalhes internos, vejamos como ele termina. Não há nenhum ret ou jmp no seu fim. Após desativar as interrupções com o cli na linha 6315, a execução continua em *_restart*. *_S_call* é a chamada de sistema correspondente do mecanismo de tratamento de interrupção. O controle chega a *_s_call* se-

guindo uma interrupção de software, isto é, a execução de uma instrução int *nnn*. As interrupções de software são tratadas como interrupções de hardware, exceto que, naturalmente, o índice na Tabela de Descritores de Interrupção é codificado na parte *nnn* de uma instrução int *nnn*, em vez de ser fornecida por um chip controlador de interrupções. Assim, quando se entrou em *_s_call*, a CPU já alternou para uma pilha dentro da tabela de processos (fornecida pelo TSS) e vários registradores já foram acrescentados a essa pilha. Caindo para *_restart*, a chamada a *_s_call* por fim termina com uma instrução iretd, e, assim como nas interrupções de hardware, essa instrução iniciará qualquer que seja o processo apontado por *proc_ptr* nesse ponto. A Figura 2-34 compara o gerenciamento de uma interrupção de hardware e de uma chamada de sistema utilizando o mecanismo de interrupção de software.

Vejamos agora alguns detalhes de *_s_call*. O rótulo alternativo, *_p_s_call*, é um vestígio da versão de 16 bits do MINIX, que tem rotinas separadas para operação no modo protegido e no modo real. Na versão de 32 bits todas as chamadas a qualquer rótulo acabam aqui. Um programador que invoca uma chamada de sistema do MINIX escreve uma chamada de função em C que se parece com qualquer outra chamada de função, seja para uma função localmente definida ou para uma rotina na biblioteca de C. O código da biblioteca que suporta uma chamada de sistema configura uma mensagem, carrega o endereço da mensagem e o id do processo de destino em registradores de CPU e, então, invoca uma instrução int *SYS386_VECTOR*. Como descrito acima, o resultado é que o controle passa para o início de *_s_call* e vários registradores já foram acrescentados a uma pilha dentro da tabela de processos.

A primeira parte do código *_s_call* assemelha-se a uma expansão inline de *save* e salva os registradores adicionais que devem ser conservados. Assim como em *save*, uma instrução

```
mov    esp, k_stktop
```

alterna, então, para a pilha do *kernel*, e as interrupções são reativadas (a semelhança de uma interrupção de software com uma interrupção de hardware estende-se a ambos desativando todas as interrupções). Seguindo, vem uma chamada a *_sys_call*, que discutiremos na próxima seção. Por enquanto, simplesmente diremos que ela faz com que uma mensagem seja enviada, e que esta última, por sua vez, faz com que o agendador seja executado. Assim, quando *_sys_call* retorna, é provável que *proc_ptr* esteja apontando para um processo diferente do que iniciou a chamada de sistema. Antes de a execução ir para *restart*, uma instrução cli desativa as interrupções para proteger a estrutura de pilha do processo que está para ser iniciado.

Vimos que *_restart* (linha 6322) é alcançado de várias maneiras:

1. Por uma chamada a partir de *main*, quando o sistema inicia.
2. Por um salto a partir de *hwint_master* ou *hwint_slave*, após uma interrupção de hardware.
3. Por meio de *_s_call*, após uma chamada de sistema.

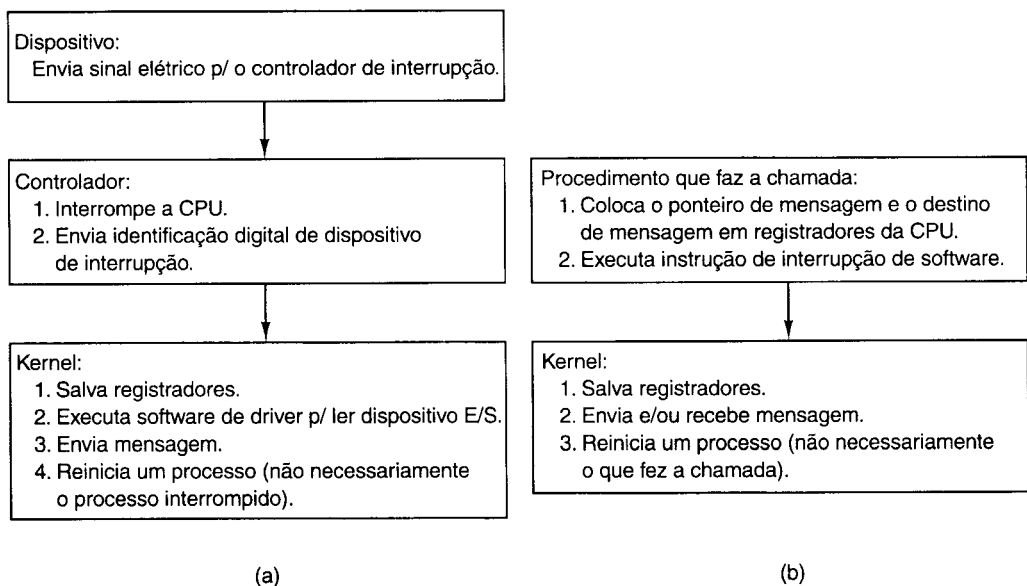



Figura 2-34 (a) Como uma interrupção de hardware é processada. (b) Como uma chamada de sistema é feita.

Em cada caso, as interrupções são desativadas neste ponto. *_Restart* chama *unhold* se detecta que qualquer interrupção não-servida foi contida porque chegou enquanto outras interrupções estavam sendo processadas. Isso permite que outra interrupção seja convertida em mensagens antes de qualquer processo ser reiniciado. Isso temporariamente reativa as interrupções, mas elas são desativadas novamente antes de *unhold* retornar. Na linha 6333, o próximo processo a executar foi definitivamente escolhido, e com as interrupções desativadas ele não pode ser alterado. A tabela de processos foi cuidadosamente construída para começar com uma estrutura de pilha, e a instrução nessa linha,

```
mov     esp, (_proc_ptr)
```

aponta o ponteiro de pilha da CPU para a estrutura da pilha. A instrução

```
lldt    P_LDT_SEL (esp)
```

carrega então o registrador da tabela descritora local do processador a partir da estrutura de pilha. Isso prepara o processador para utilizar os segmentos de memória que pertencem ao próximo processo a ser executado. A instrução seguinte carrega o endereço na entrada de tabela de processos do próximo processo, que é onde a pilha para a próxima interrupção será configurada, e a instrução seguinte armazena esse endereço no TSS. A primeira parte de *_restart* não é necessária depois que uma interrupção ocorrida quando o código do *kernel* (incluindo o código do serviço de interrupção) estava executando, uma vez que a pilha do *kernel* estará em utilização, e o término do serviço de interrupção permite que o código do *kernel* continue. O rótulo *restart1* (linha 6337) marca o ponto onde a execução é reassumida neste caso. Neste ponto *k_reenter* é decrementado para registrar o nível de interrupções possivelmente aninhadas que foram descartadas, e as instruções restantes restauram o processador para o estado em que ele estava quando o próximo processo executou pela última vez. A penúltima instrução modifica o ponteiro da pilha de tal modo que o endereço de retorno que foi colocado na pilha quando *save* foi chamada é ignorado. Se a última interrupção ocorreu quando um processo estava executando, a instrução final, iretd, completa o retorno à execução de qualquer que seja o processo que tenha permissão para executar em seguida, restaurando seus registradores restantes, incluindo seu segmento de pilha e o ponteiro de pilha. Se, porém, esse encontro com o iretd ocorreu via *restart1*, a pilha em utilização pelo *kernel* não é uma estrutura de pilha, mas sim a pilha do *kernel*, e isso não é um retorno para um processo interrompido, mas sim o término de uma interrupção que ocorreu enquanto o código do *kernel* estava executando. A CPU detecta isso quando o descritor de segmento de código é retirado da pilha durante execução de iretd, e a ação completa de iretd neste caso é manter a pilha do *kernel* em utilização.

Há mais algumas coisas a discutir sobre *mpx386.s*. Além das interrupções de hardware e de software, várias condições de erro internas à CPU podem causar uma **exceção.** As exceções não são sempre ruins. Elas podem ser utilizadas para estimular o sistema operacional a oferecer um serviço, como fornecer mais memória para um processo utilizar, ou para carregar de uma página de memória que foi movida para o disco, embora tais serviços não sejam implementados no MINIX-padrão. Mas, quando ocorre uma exceção, ela não deve ser ignorada. As exceções são gerenciadas pelo mesmo mecanismo que as interrupções, utilizando descritores na Tabela de Descritores de Interrupção. Essas entradas na tabela apontam para os 16 pontos de entrada de manipuladores de exceção, começando com *_divide_error* e terminando com *_copr_error*, localizados perto do final de *mpx386.s*, nas linhas 6350 a 6412. Todos esses saltam para *exception* (linha 6420) ou *errexception* (linha 6431) dependendo de se a condição acrescenta um código de erro à pilha ou não. O gerenciamento aqui, no código *assembly*, é semelhante ao que nós já vimos. Os registradores são colocados na pilha, e a rotina em C *_exception* (note o sublinhado) é chamada para gerenciar o evento. As conseqüências das exceções variam. Algumas são ignoradas, algumas causam pânico e algumas resultam no envio de sinais aos processos. Examinaremos *_exception* em uma seção posterior.

Há um outro ponto de entrada que é gerenciado como uma interrupção, *_level0_ call* (linha 6458). Sua função será discutida na próxima seção, quando discutiremos o código para o qual ele salta, *_level0_func*. O ponto de entrada está aqui em *mpx386.s* junto com os pontos de entrada de interrupção e de exceção porque ele também é invocado pela execução de uma instrução int. Como as rotinas de exceção, ele chama *save* e, assim, por fim o código que salta para cá terminará com um ret que conduz a *_restart*. A última função executável em *mpx386.s* é *_idle_task* (linha 6465). Essa é um laço que não faz nada e é executada sempre que não há outro processo pronto para executar.

Finalmente, algum espaço de armazenamento de dados é reservado no fim do arquivo de linguagem *assembly*. Há dois diferentes segmentos de dados definidos aqui. A declaração

```
.sect .rom
```

na linha 6478 assegura que esse espaço de armazenamento é alocado bem no começo do segmento de dados do *kernel*. O compilador põe um número mágico aqui para que *boot* possa verificar se o arquivo que ele carrega é uma imagem válida do *kernel*. *Boot*, então, sobrescreve o número mágico e o espaço subseqüente com a matriz de dados *_sizes*, como descrito na discussão sobre estruturas de dados do *kernel*. Espaço suficiente é reservado para uma matriz *_sizes* com um total de 16 entradas, no caso de ser-

vidores adicionais serem adicionados ao MINIX. A outra área de armazenamento de dados definida na declaração

```
.sect .bss
```

(linha 6483) reserva espaço na área normal de variáveis não-inicializadas do *kernel* para a pilha do *kernel* e para variáveis utilizadas pelos manipuladores de exceção. Servidores e processos normais têm espaço de pilha reservado quando um arquivo executável é vinculado e dependem do *kernel* para configurar adequadamente o descritor de segmento de pilha e o ponteiro de pilha quando são executados. O *kernel* tem de fazer isso sozinho.

2.6.8 Comunicação Interprocesso no MINIX

Os processos no MINIX comunicam-se por mensagens, utilizando o princípio do *rendez-vous*. Quando um processo faz um SEND, a camada mais baixa do *kernel* verifica se o destino está esperando uma mensagem do remetente (ou de qualquer remetente). Se estiver, a mensagem é copiada do buffer do remetente para o buffer do destinatário, e os dois processos são marcados como executáveis. Se o destino não estiver esperando uma mensagem do remetente, este último é marcado como bloqueado e colocado em uma fila de processos esperando poder enviar para o destinatário.

Quando um processo faz um RECEIVE, o *kernel* verifica se qualquer processo está enfileirado tentando enviar para ele. Se estiver, a mensagem é copiada do remetente bloqueado para o destinatário, e ambos são marcados como executáveis. Se nenhum processo estiver enfileirado tentando enviar para ele, o destinatário bloqueia até a chegada de uma mensagem.

O código de alto nível para comunicação interprocesso está localizado em *proc.c*. O trabalho do *kernel* é traduzir uma interrupção de hardware ou uma interrupção de software em uma mensagem. A primeira é gerada por hardware e a última é a maneira como uma requisição de serviços do sistema, isto é, uma chamada de sistema, é comunicada ao *kernel*. Esses casos são tão semelhantes que poderiam ser gerenciados por uma única função, mas foi mais eficiente criar duas funções especializadas.

Primeiro veremos *interrupt* (linha 6938). Essa é chamada pela rotina do serviço de interrupção de baixo nível para um dispositivo depois de receber uma interrupção de hardware. A função de *interrupt* é converter a interrupção em uma mensagem para a tarefa que gerencia o dispositivo que gerou interrupção, e geralmente muito pouco é feito antes de chamar *interrupt*. Por exemplo, todo manipulador de interrupção de baixo nível para o *driver* de disco rígido consiste nessas três linhas tão-somente:

```
w_status = in_byte(w_wn->base + REG_STATUS); /*
   reconhecimento da interrupção */
interrupt(WINCHESTER);
return 1;
```

Se não fosse necessário ler uma porta de E/S no controlador de disco rígido para obter o status, a chamada a *interrupt* poderia ficar em *mpx386.s* em vez de em *at_wini.c*. A primeira coisa que *interrupt* faz é verificar se uma interrupção já estava sendo servida quando a interrupção atual foi recebida, olhando na variável *k_reenter* (linha 6962). Nesse caso, a interrupção atual é enfileirada, e *interrupt* retorna. A interrupção atual será servida mais adiante, quando *unhold* for chamado. A próxima ação é verificar se a tarefa está esperando uma interrupção (linhas 6978 a 6981). Se a tarefa não estiver pronta para receber, seu sinalizador *p_int_blocked* é ligado — veremos mais adiante que isso possibilita recuperar a interrupção perdida — e nenhuma mensagem é enviada. Se passar nesse teste, a mensagem é enviada. Enviar uma mensagem de HARDWARE para uma tarefa é simples, porque as tarefas e o *kernel* são compiladas no mesmo arquivo e podem acessar as mesmas áreas de dados. O código nas linhas 6989 a 6992 envia a mensagem, preenchendo os campos de origem e tipo do buffer de mensagem da tarefa de destino, redefinindo o sinalizador *RECEIVING* do destino e desbloqueando a tarefa. Uma vez que a mensagem está pronta, a tarefa de destino é agendada para executar. Discutiremos agendamento em mais detalhe na próxima seção, mas o código de *interrupt* nas linhas 6997 a 7003 fornece uma visualização do que veremos — é um substituto *inline* para o procedimento *ready* que é chamado para enfileirar um processo. Ele é simples aqui, uma vez que as mensagens originárias de interrupções vão só para tarefas e, portanto, não há nenhuma necessidade de determinar qual das três filas de processos precisa ser alterada.

A próxima função em *proc.c* é *sys_call*. Ela tem uma função semelhante a *interrupt*: converte uma interrupção de software (a instrução int *SYS386_VECTOR* por meio da qual uma chamada de sistema é iniciada) em uma mensagem. Mas como há um intervalo maior de possíveis origens e de destinos nesse caso, e como a chamada pode requerer que se envie e/ou receba uma mensagem, *sys_call* tem mais trabalho a fazer. Como é freqüentemente o caso, isso significa que o código para *sys_call* é curto e simples, já que faz a maioria do seu trabalho chamando outros procedimentos. A primeira dessas chamadas é para *isoksrc_dest*, uma macro definida em *proc.h* (linha 5172), que incorpora ainda outra macro, *isokprocn*, também definida em *proc.h* (linha 5171). O efeito é verificar se o processo especificado como a origem ou o destino da mensagem é válido. Na linha 7026 um teste semelhante, *isuserp* (também uma macro definida em *proc.h*), é executado para garantir que, se a chamada é de um processo de usuário, ele deve estar solicitando o envio de uma mensagem e, então, aguardando uma resposta, o único tipo de chamada permitido a processos de usuário. Esse tipo de erro é improvável, mas os testes são facilmente feitos, já que acabam sendo compilados no código para executar comparações de números inteiros pequenos. Nesse nível mais básico do sistema operacional, é aconselhável fazer testes para

os erros mais improváveis. Esse código pode executar muitas vezes por segundo a cada segundo de atividade do sistema de computador em que ele estiver executando.

Por fim, se a chamada requer enviar uma mensagem, *mini_send* é chamada (linha 7031), e se for exigido receber uma mensagem, *mini_rec* é chamada (linha 7039). Essas funções são o coração do mecanismo normal de passagem de mensagens do MINIX e merecem estudo cuidadoso.

Mini_send (linha 7045) tem três parâmetros: o processo que fez a chamada, o processo para o qual enviar e um ponteiro para o buffer onde a mensagem está. Ela executa vários testes. Primeiro, certifica-se de que os processos de usuário tentam enviar mensagens apenas para FS ou para MM. Na linha 7060, o parâmetro *caller_ptr* é testado com a macro *isuserp* para determinar se o processo que fez a chamada é um processo de usuário, e o parâmetro *dest* é testado com uma função similar, *issysentn*, para determinar se é FS ou MM. Se a combinação não for permitida, *mini_send* termina com um erro.

Em seguida, é feita uma verificação para assegurar que o destino da mensagem é um processo ativo, não uma entrada vazia na tabela de processos (linha 7062). Nas linhas 7068 a 7073, *mini_send* verifica se a mensagem cabe inteiramente dentro do segmento de dados do usuário, do segmento de código ou do intervalo entre eles. Se não, um código de erro é retornado

O próximo teste é verificar um possível impasse. Na linha 7079 há um teste para assegurar que o destino da mensagem não está tentando enviar uma mensagem de volta ao que fez a chamada.

O teste-chave em *mini_send* está nas linhas 7088 a 7090. Aqui uma verificação é feita para ver se o destino está bloqueado em um RECEIVE, como mostrado pelo bit *RECEIVING* no campo *p_flags* de sua entrada da tabela de processos. Se estiver esperando, então, a próxima pergunta é: quem ele está esperando?" Se estiver esperando o remetente, ou qualquer outro, *CopyMess* é executado para copiar a mensagem, e o destinatário é desbloqueado redefinindo seu bit *RECEIVING*. *CopyMess* é definida como uma macro na linha 6932. Ela chama a rotina de linguagem *assembly cp_mess* em *klib386.s*.

Se, por outro lado, o destinatário não estiver bloqueado, ou estiver bloqueado, mas esperando uma mensagem de outro, o código nas linhas 7098 a 7111 é executado para bloquear e enfileirar o remetente. Todos os processos que estão querendo enviar para um determinado destino são enfileirados juntos em uma lista encadeada, com o campo *p_callerq* do destino apontando para entrada da tabela de processos do processo na cabeça da fila. O exemplo da Figura 2-35(a) mostra o que acontece quando o processo 3 é incapaz de enviar para o processo 0. Se subseqüentemente o processo 4 também for incapaz de enviar para o processo 0, obteremos a situação da Figura 2-35(b).

Mini_rec (linha 6119) é chamada por *sys_call* quando seu parâmetro *function* é *RECEIVE* ou *BOTH*. O laço nas linhas 7137 a 7151 pesquisa os processos enfileirados que estão esperando enviar para o receptor para ver se qualquer um deles é aceitável. Se encontrar algum, a mensagem é copiada do remetente para o receptor; então, o remetente é desbloqueado, tornado pronto para executar e removido da fila de processos que tentam enviar para o receptor.

Se nenhum remetente conveniente for encontrado, uma verificação é feita para ver se o sinalizador *p_int_blocked* do processo do destinatário indica que uma interrupção para esse destino foi anteriormente bloqueada (linha 7154). Se foi, uma mensagem é criada neste ponto — uma vez que as mensagens de *HARDWARE* não têm nenhum outro conteúdo além de *HARDWARE* no campo de origem e *HARD_INT* no campo de tipo, não há nenhuma necessidade de chamar *CopyMess* neste caso.

Se não for encontrada uma interrupção bloqueada, a fonte e o endereço do buffer do processo são salvos em sua entrada na tabela de processos e ele é marcado como bloqueado ligando-se seu bit *RECEIVING*. A chamada a *unready* na linha 7165 remove o destinatário da fila de processos executáveis do agendador. A chamada é condicional para evitar o bloqueio do processo se ainda houvesse outro bit ligado em seus *p_flags*; um sinal pode estar pendente, e

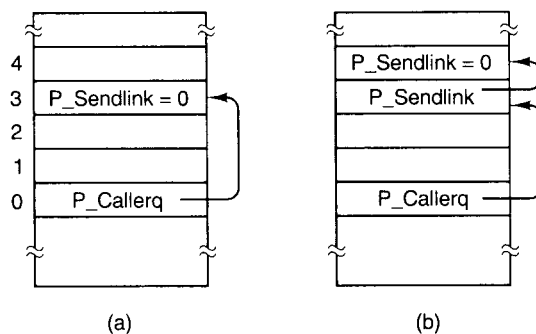



Figura 2-35 O enfileiramento de processos que tentam enviar para o processo 0.

o processo deve ter outra chance de executar logo para tratar do sinal.

A penúltima declaração em *mini_rec* (linhas 7171 e 7172) está relacionada com o modo como os sinais gerados pelo *kernel SIGINT, SIGQUIT* e *SIGALRM* são gerenciados. Quando um desses sinais ocorre, uma mensagem é enviada para o gerenciador de memória, se ele estiver esperando uma mensagem de ANY. Se não, o sinal é memorizado no *kernel* até que o gerenciador de memória tente, por fim, receber de ANY. Isso é testado aqui, e, se necessário, *inform* é chamado para informá-lo dos sinais pendentes.

2.6.9 Agendamento no MINIX

O MINIX utiliza um algoritmo de agendamento de múltiplos níveis que segue bem a estrutura mostrada na Figura 2-26. Nessa figura, vemos tarefas de E/S na camada 2, processos de servidor na camada 3 e processos de usuário na camada 4. O agendador mantém três filas de processos executáveis, um para cada camada, como mostrado na Figura 2-36. A matriz *rdy_head* tem uma entrada para cada fila, com essa entrada apontando para o processo na cabeça da fila. De maneira semelhante, *rdy_tail* é uma matriz cujas entradas apontam para o último processo em cada fila. Essas duas matrizes são definidas com a macro *EXTERN* em *proc.h* (linhas 5192 e 5193).

Sempre que um processo bloqueado é acordado, ele é anexado ao fim de sua fila. A existência da matriz *rdy_tail* torna a adicionar um processo ao fim de uma fila uma definição eficiente. Sempre que um processo em execução torna-se bloqueado, ou um processo executável é eliminado por um sinal, esse processo é removido das filas do agendador. Somente processos executáveis são enfileirados.

Dadas as estruturas de fila que acabamos de descrever, o algoritmo de agendamento é simples: encontrar a fila de prioridade mais alta que não está vazia e selecionar o processo na cabeça dessa fila. Se todas as filas estiverem vazias, a rotina de espera[1] será executada. Na Figura 2-36, *TASK_Q* tem a maior prioridade. O código de agendamento está em *proc.c*. A fila é escolhida em *pick_proc* (linha 7179). O principal trabalho dessa função é configurar *proc_ptr*. Qualquer alteração nas filas, que possa afetar a escolha de qual processo deve executar em seguida, requer que *pick_proc* seja chamada novamente. Sempre que o processo atual bloqueia, *pick_proc* é chamada para reagendar a CPU.

Pick_proc é simples. Há um teste para cada fila. *TASK_Q* é testada primeiro e se um processo nessa fila estiver pronto, *pick_proc* configura *proc_ptr* e imediatamente retorna. Em seguida, *SERVER_Q* é testada, e, novamente, se um processo estiver pronto, *pick_proc* configura *proc_ptr* e retorna. Se houver um processo pronto na fila *USER_Q*, *bill_ptr* é alterada para contabilizar ao processo de usuário o tempo da CPU que está para ser fornecido (linha 7198). Isso garante que o último processo de usuário a executar seja cobrado pelo trabalho feito em seu benefício pelo sistema. Se nenhuma das filas tem uma tarefa pronta, a linha 7204 transfere a cobrança para o processo *IDLE* e o agenda. O processo escolhido para executar não é removido de sua fila meramente porque foi selecionado.

Os procedimentos *ready* (linha 7210) e *unready* (linha 7258) são chamados para colocar um processo executável em sua fila e remover um processo não mais executável de sua fila, respectivamente. *Ready* é chamado tanto a partir de *mini_send* como de *mini_rec,* como vimos. Ele também poderia ter sido chamado a partir de *interrupts* mas, a fim de acelerar o processamento da interrupção, seu equivalente funcional foi escrito como parte do código de *interrupt*. *Ready* manipula uma das três filas de processos. Ele simplesmente adiciona o processo ao final da fila apropriada.

Unready também gerencia as filas. Normalmente, o processo que ele remove está na cabeça da sua fila, já que um processo deve estar executando para ser bloqueado. Nesse caso, *unready* chama *pick_proc* antes de retornar, como, por exemplo, na linha 7293. Um processo de usuário que não está executando também pode tornar-se não-pronto se ele enviar um sinal e se o processo não for encontrado na cabeça de uma das filas, uma pesquisa é feita ao longo de *USER_Q* e ele é removido se for encontrado.

Embora a maioria das decisões de agendamento seja feita quando um processo bloqueia ou desbloqueia, o agendamento também deve ser feito quando a tarefa de relógio nota que o processo de usuário atual excedeu seu *quantum*. Nesse caso, a tarefa de relógio chama *sched* (linha 7311) para mover o processo na cabeça de *USER_Q* para o fim dessa fila. Esse algoritmo resulta na execução de processos de usuário estritamente no estilo *round robin*. O

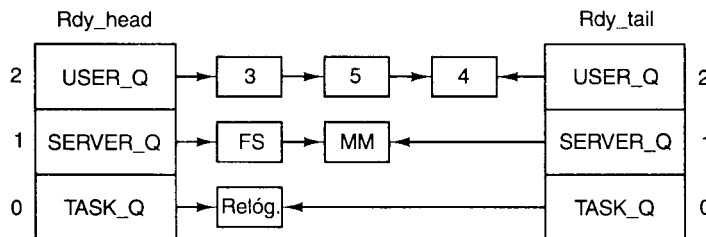



Figura 2-36 O agendador mantém três filas, uma por nível de prioridade.

sistema de arquivos, o gerenciador de memória e as tarefas de E/S nunca são colocados no fim de suas filas porque estiveram executando por muito tempo. Confia-se em que elas funcionarão adequadamente e bloquearão depois de finalizarem seu trabalho.

Ainda há mais algumas rotinas em *proc.c* que suportam agendamento de processos. Cinco destas, *lock_mini_send, lock_pick_proc, lock_ready, lock_unready e lock_sched,* definem um bloqueio, utilizando a variável *switching* antes de chamar a função correspondente e, então, liberam o bloqueio ao seu término. A última função neste arquivo, *unhold* (linha 7400), foi mencionada em nossa discussão sobre *_restart* em *mpx386.s.* Ela circula pela fila de interrupções suspensas, chamando *interrupt* para cada uma, a fim de ter todas as interrupções pendentes convertidas em mensagens antes de outro processo ter permissão para executar.

Em resumo, o algoritmo de agendamento mantém três filas de prioridade, uma para as tarefas de E/S, uma para os processos de servidor e uma para os processos de usuário. O primeiro processo na fila de prioridade mais alta sempre é executado primeiro. As tarefas e os servidores sempre têm permissão para executar até bloquearem, mas a tarefa de relógio monitora o tempo utilizado por processos de usuário. Se um processo de usuário utilizar todo seu quantum, ele é colocado no fim de sua fila, obtendo, assim, um agendamento por *round robin* simples entre os processos de usuário.

2.6.10 Suporte de *Kernel* Dependente de Hardware

Há várias funções de C que são muito dependentes do hardware. Para facilitar portar o MINIX para outros sistemas, essas funções foram separadas em arquivos que serão discutidos nessa seção, *exception.c, i8259.c* e *protect.c.* em vez de serem incluídas nos mesmos arquivos com o código de nível mais alto que elas suportam.

Exception.c contém o manipulador de exceções, *exception* (linha 7512), que é chamado (como *_exception)* pela parte de linguagem *assembly* do código de manipulação de exceções em *mpx386.s.* As exceções que se originam de processos de usuário são convertidas em sinais. É aceitável, e até provável, que os usuários cometam erros em seus programas, mas uma exceção que se origina no sistema operacional indica que algo está seriamente errado e causa pânico. A matriz *ex_data* (linhas 7522 a 7540) determina a mensagem de erro a ser impressa em caso de pânico, ou o sinal a ser enviado para um processo de usuário para cada exceção. Os processadores Intel antigos não geram todas as exceções, e o terceiro campo em cada entrada indica o modelo mínimo de processador que é capaz de gerar cada uma. Essa matriz fornece um resumo interessante da evolução da família de processadores Intel sobre a qual o MINIX foi implementado. Na linha 7563, uma mensagem alternativa é impressa se um pânico resulta de uma interrupção que não seria esperada do processador em uso.

As três funções em *i8259.c* são utilizadas durante a inicialização do sistema para inicializar os chips controladores de interrupções Intel 8259. *Intr_init* (linha 7621) inicializa os controladores. Ela grava dados em várias localizações de porta. Em algumas poucas linhas, uma variável derivada dos parâmetros de *boot* é testada, por exemplo, a primeira porta escreve na linha 7637, para acomodar diferentes modelos de computador. Na linha 7638 e novamente na linha 7644, o parâmetro *mine* é testado e um valor apropriado para o MINIX ou para o BIOS ROM é gravado na porta. Ao sair do MINIX, *intr_init* pode ser chamada para restaurar os vetores de BIOS, oferecendo uma saída elegante para voltar ao monitor de *boot*. *Mine* seleciona o modo a utilizar. O pleno entendimento do que está acontecendo aqui exigiria estudo da documentação do circuito integrado 8259 e assim não nos demoraremos nos detalhes. Indicaremos apenas que a chamada *out_byte* na linha 7642 faz o controlador-mestre ignorar qualquer entrada exceto a do escravo, e a operação semelhante na linha 7648 inibe a resposta do escravo para todas as suas entradas. Além disso, a linha final da função pré-carrega o endereço de *spurious_irq*, a próxima função no arquivo (linha 7657), em cada entrada em *irq_table.* Isso assegura que qualquer interrupção gerada antes de os manipuladores reais serem instalados não causará nenhum dano.

A última função em *i8259.c* é *put_irq_manipulador* (linha 7673). Na inicialização, cada tarefa que deve responder a uma interrupção chama essa função para colocar seu próprio endereço de manipulador na tabela de interrupção, sobrescrevendo o endereço de *spurious_irq*.

Protect.c contém rotinas relacionadas com a operação no modo protegido dos processadores Intel. A **Tabela Global de Descritores** (*Global Descriptor Table*, GDT), as **Tabelas Locais de Descritores** (*Local Descriptor Tables*, LDTs) e a **Tabela de Descritores de Interrupção** (*Interrupt Descriptor Table*, IDT), todas localizadas na memória, fornecem acesso protegido para recursos do sistema. A GDT e a IDT são apontadas por registradores especiais dentro da CPU, e as entradas GDT apontam para LDTs. A GDT está disponível para todos os processos e armazena descritores de segmento para regiões de memória utilizadas pelo sistema operacional. Há normalmente uma LDT para cada processo, que armazena descritores de segmento para as regiões de memória utilizadas pelo processo. Os descritores são estruturas de 8 bytes com vários componentes, mas as partes mais importantes de um descritor de segmento são os campos que descrevem o endereço de base e o limite de uma região da memória. A IDT também é composta de descritores de 8 bytes, com a parte mais importante sendo o endereço do código a ser executado quando a correspondente interrupção é ativada.

Prot_init (linha 7767) é chamado por *start.c* para configurar a GDT nas linhas 7828 a 7845. O BIOS do IBM PC requer que ele seja solicitado de uma certa maneira e todos os índices para ele definidos em *protect.h.* O espaço para uma LDT para cada processo é alocado na tabela de processos. Cada um contém dois descritores, um para o seg-

mento de código e um para o segmento de dados — lembre-se de que estamos discutindo aqui os segmentos como definidos pelo hardware; que não são os mesmos segmentos gerenciados pelo sistema operacional, o qual considera o segmento de dados definido por hardware como dividido ainda nos segmentos de dados e de pilha. Nas linhas 7851 a 7857, são construídos descritores para cada LDT na GDT. As funções *init_dataseg* e *init_codeseg* realmente criam tais descritores. As entradas nas próprias LDTs são iniciadas quando um mapa de memória de processo é alterado (i. e., quando uma chamada de sistema EXEC é feita).

Outra estrutura de dados do processador que precisa de inicialização é o **Segmento de Estado de Tarefa** (*Task State Segment*, TSS). A estrutura é definida no início desse arquivo (linhas 7725 a 7753) e fornece espaço para armazenamento de registradores do processador e de outras informações que devem ser salvas quando uma comutação de tarefas é feita. O MINIX utiliza apenas os campos que definem onde uma nova pilha será criada quando uma interrupção ocorrer. A chamada a *init_dataseg* na linha 7867 assegura que ele pode ser localizado utilizando a GDT.

Para entender como o MINIX funciona no nível mais baixo, talvez a coisa mais importante seja entender como exceções, interrupções de hardware ou as instruções int <*nnn*> levam à execução dos vários trechos do código que foram escritos para servi-los. Isso é realizado por meio da tabela descritora de portão de interrupção. A matriz *gate_table* (linhas 7786 a 7818) é inicializada pelo compilador com os endereços das rotinas que gerenciam exceções e interrupções de hardware e, então, é utilizada no laço das linhas 7873 a 7877 para iniciar uma parte grande dessa tabela, utilizando chamadas à função *int_gate*. Os vetores restantes, *SYS_VECTOR*, *SYS386_VECTOR* e *LEVEL0_VECTOR*, exigem diferentes níveis de privilégio e são inicializados depois do laço.

Há boas razões para o modo como os dados são estruturados nos descritores, com base nos detalhes do hardware e na necessidade de manter compatibilidade entre processadores avançados e o processador 286 de 16 bits. Felizmente, em geral, podemos deixar esses detalhes para os projetistas de processadores da Intel. Na sua maior parte, a linguagem C permite evitar os detalhes. Entretanto, ao implementar um sistema operacional real os detalhes devem ser tratados em certa medida. A Figura 2-37 mostra a estrutura interna de um tipo de descritor de segmento. Note que o endereço de base a que os programas em C podem referir-se como um simples inteiro sem sinal de 32 bits, é dividido em três partes, duas das quais são separadas em porções de 1, 2 e 4 bits. O limite é uma quantidade de 20 bits armazenados como blocos separados de 4 e 16 bits. O limite é interpretado tanto como um número de bytes ou um número de páginas de 4096 bytes, baseado no valor do bit G (granulanidade). Outros descritores, como os usados para especificar como as interrupções são manipuladas, têm estruturas diferentes, mas igualmente complexas. Essas estruturas serão discutidas com mais detalhes no Capítulo 4.

A maioria das outras funções definidas em *protect.c* é dedicada a conversão entre variáveis utilizadas em programas em C e as horríveis formas que esses dados assumem nos descritores da máquina como o da Figura 2-37. *Init_codeseg* (linha 7889) e *init_dataseg* (linha 7906) são semelhantes em operação e utilizados para converter os parâmetros passados para eles para a forma de descritores de segmento. Cada um deles, por sua vez, chama a próxima função, *sdesc* (linha 7922), para completar o trabalho. Aqui é onde os confusos detalhes da estrutura mostrada na Figura 2-37 são tratados. *Init_codeseg* e *init_data_seg* não são utilizadas apenas na inicialização do sistema. Além disso, elas também são chamadas pela tarefa de sistema sempre que um novo processo é iniciado para alocar os segmentos adequados de memória para uso do processo. *Seg2phys* (linha 7947), chamado apenas a partir de *start.c*, executa uma operação inversa à operação de *sdesc*, extraindo o endereço base de um segmento a partir de um descritor de segmento. *Int_gate* (linha 7969) executa uma função semelhante às funções de *init_codeseg* e *init_dataseg* na criação de entradas da tabela de descritores de interrupções.

A função final em *protect.c*, *enable_iop* (linha 7988) faz um truque "sujo". Apontamos em vários lugares que uma das funções de um sistema operacional é proteger recursos de sistema, e de certa maneira o MINIX faz isso utilizando níveis de privilégio para certificar-se de que certas instruções fiquem fora do alcance de programas de usuário. Entretanto, o MINIX também se destina a ser executado em sistemas pequenos, que geralmente têm só um usuário ou talvez somente alguns usuários com direitos atribuídos. Em um sistema assim, um usuário poderia muito bem querer escrever um programa que acessasse portas de E/S, por exemplo, para utilizar na aquisição de dados científi-

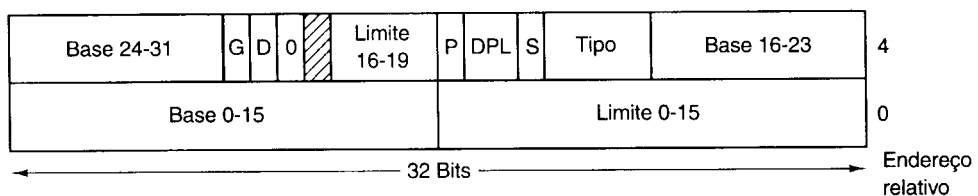



Figura 2-37 O formato de um descritor de segmento Intel.

cos. O sistema de arquivos tem um pequeno segredo construído nele — quando os arquivos */dev/mem* ou */dev/kmem* são abertos, a tarefa de memória chama *enable_iop*, que altera o nível de privilégio para operações de E/S, permitindo que o processo atual execute instruções que lêem e gravam em portas de E/S. A descrição do propósito da função é mais complicada que a função em si, que apenas configura 2 bits, na palavra da estrutura de pilha na entrada do processo de chamada, que serão carregados no registrador de status da CPU, quando o processo seguinte for executado. Não há nenhuma necessidade de outra função desfazer isso, uma vez que se aplicará apenas ao processo de chamada.

2.6.11 Utilitários e Biblioteca do *Kernel*

Por fim, o *kernel* tem uma biblioteca de funções de suporte escritas em linguagem *assembly* que são incluídas compilando *klib.s* e alguns programas utilitários, escritos em C, no arquivo *misc.c*. Vejamos primeiro os arquivos de linguagem *assembly*. *Klib.s* (linha 8000) é um arquivo curto semelhante a *mpx.s*, que seleciona a versão específica de máquina apropriada com base na definição de *WORD_SIZE*. O código que discutiremos está em *klib386.s* (linha 8100). Ele contém aproximadamente duas dúzias de rotinas utilitárias que estão em código *assembly*, seja por eficiência ou porque elas absolutamente não podem ser escritas em C.

_Monitor (linha 8166) possibilita retornar para o monitor de *boot*. Do ponto de vista do monitor de *boot*, todo o MINIX é apenas uma sub-rotina e quando o MINIX é iniciado, um endereço de retorno para o monitor é deixado na pilha do monitor. *_Monitor* precisa apenas restaurar os vários seletores de segmento e o ponteiro de pilha que foi salvo quando o MINIX foi iniciado, e, então, retornar como de qualquer outra sub-rotina.

A próxima função, *_check_mem* (linha 8198), é utilizada no momento da inicialização para determinar o tamanho de um bloco de memória. Ela executa um teste simples em cada 16° byte, utilizando dois padrões que testam cada bit com os valores "0" e "1".

Embora *_phys_copy* (veja mais adiante) possa ter sido utilizado para copiar mensagens, *_cp_mess* (linha 8243), um procedimento especializado mais rápido, foi fornecido para esse propósito. Ele é chamado por

```
cp_mess(source, src_clicks, src_offset, dest_clicks,
dest_offset);
```

onde *source* é o número de processo do remetente, o qual é copiado no campo *m_source* do buffer do destinatário. Os endereços da origem e do destino são ambos especificados fornecendo um número *click*, em geral, a base do segmento que contém o buffer, e um deslocamento (*offset*) desse *click*. Essa forma de especificar a origem e o destino é mais eficiente que os endereços de 32 bits utilizados por *phys_copy*

Exit, * _exit* e *_ _ _exit* (linhas 8283 a 8285) são definidos porque algumas rotinas de biblioteca que podem ser utilizadas ao compilar o MINIX fazem chamadas à função-padrão *exit* do C. Uma saída do *kernel* não é um conceito que faz sentido; não há nenhum lugar para onde ir. A solução aqui é ativar as interrupções e entrar em um laço interminável. Eventualmente, uma operação de E/S ou o relógio acabarão causando uma interrupção, e a operação normal do sistema reassumirá. O ponto de entrada para *_ _ _main* (linha 8289) é outra tentativa de lidar com uma ação de compilador que, apesar de poder fazer sentido ao compilar um programa de usuário, não tem qualquer propósito no *kernel*. Ela aponta para uma instrução ret de linguagem *assembly* (retorno de sub-rotina).

_In_byte (linha 8300), *_in_word* (linha 8314), *_out_byte* (linha 8328) e *_out_word* (linha 8342) fornecem acesso a portas de E/S, que no hardware Intel ocupam um espaço separado de endereçamento da memória e utilizam instruções diferentes para leitura e para gravação de memória. *_Port_read* (linha 8359), *_port_read_byte* (linha 8386), *_port_write* (linha 8412) e *_port_write_byte* (linha 8439) gerenciam a transferência de blocos de dados entre portas de E/S e de memória; eles são utilizados principalmente para transferências para e a partir do disco, que devem ser feitas mais rapidamente do que é possível com outras chamadas de E/S. As versões de byte lêem 8 bits em vez de 16 em cada operação para acomodar dispositivos periféricos de 8 bits mais antigos.

Ocasionalmente, uma tarefa precisa desativar todas interrupções de CPU temporariamente. Ela faz isso chamando *_lock* (linha 8462). Quando as interrupções puderem ser reativadas, a tarefa pode chamar *_unlock* (linha 8474) para ativar as interrupções. Uma única instrução de máquina executa cada uma dessas operações. Em contraste, o código para *_Enable_irq* (linha 8488) e *_disable_irq* (linha 8521) é mais complicado. Eles funcionam no nível do chip controlador de interrupção para ativar e desativar interrupções de hardware individualmente.

_Phys_copy (linha 8564) é chamada em C por

```
phys_copy (source_address, destination_address,
bytes);
```

e copia um bloco de dados em qualquer lugar na memória física para qualquer outro lugar. Os dois endereços são absolutos, isto é, endereço 0 significa realmente o primeiro byte no espaço inteiro de endereçamento, e todos os três parâmetros são números inteiros longos sem sinal (*unsigned longs*).

As duas curtas funções seguintes são muito específicas de processadores Intel. *_Mem_rdw* (linha 8608) retorna uma palavra de 16 bits de qualquer lugar na memória. O resultado é estendido com zero no registrador de 32 bits *eax*. A função *_reset* (linha 8623) reinicia o processador. Ela faz isso carregando o registrador da tabela de descritores de interrupções do processador com um ponteiro nulo e, então, executa uma interrupção de software. Isso tem o mesmo efeito que desligar e ligar o hardware.

As próximas duas rotinas suportam o monitor de vídeo e são utilizadas pela tarefa de console. *_Mem_vid_copy* (linha 8643) copia uma *string* de palavras que contém bytes de caractere e de atributo alternados, da região da memória do *kernel* para a memória do monitor de vídeo. *_Vid_vid_copy* (linha 8696) copia um bloco dentro da própria memória de vídeo. Isso é um pouco mais complicado, uma vez que o bloco de destino pode sobrepor-se ao bloco de origem, e a direção do movimento é importante.

A última função nesse arquivo é *_level0* (linha 8773). Ela permite que as tarefas tenham o nível de permissão de maior privilégio, o nível zero, quando necessário. Ela é utilizada para coisas como reiniciar a CPU ou acessar as rotinas de ROM BIOS do PC.

Os utilitários de linguagem C em *misc.c* são especializados. *Mem_init* (linha 8820) é chamado só por *main*, quando o MINIX é inicializado. Pode haver duas ou três regiões disjuntas de memória em um computador compatível com IBM-PC. O tamanho do intervalo mais baixo, conhecido pelos usuários de PC como memória "baixa", e do intervalo de memória que inicia acima da área de ROM do PC (memória "estendida") são informadas pelo BIOS ao monitor de *boot*, que, por sua vez, passa os valores como parâmetros de *boot*, os quais são interpretados por *cstart* e gravados em *low_memsize* e *ext_memsize* no momento do *boot*. A terceira região é memória de "sombra", na qual a ROM BIOS pode ser copiada para oferecer uma melhora no desempenho, já que a memória ROM é normalmente mais lenta que a memória gravável. Como o MINIX normalmente não utiliza o BIOS, *mem_init* tenta localizar essa memória e adicioná-la à memória disponível para sua utilização. Ele faz isso chamando *check_mem* para testar a região de memória onde essa memória, às vezes, pode ser encontrada.

A próxima rotina, *env_parse* (linha 8865), também é utilizada no momento da inicialização. O monitor de *boot* pode passar *strings* arbitrárias como "DPETH0=300:10" ao MINIX nos parâmetros do *boot*. *Env_parse* tenta localizar uma *string* cujo primeiro campo coincida com seu primeiro argumento, *env*, e, então, extrai o campo requerido. Os comentários no código explicam a utilização da função. Ela é fornecida principalmente para ajudar o usuário que quer adicionar novos *drivers* que podem precisar receber parâmetros. O exemplo "DPETH0" é utilizado para passar informações de configuração a um adaptador Ethernet quando o suporte de rede é compilado no MINIX.

As duas últimas rotinas que discutiremos neste capítulo são *bad_assertion* (linha 8935) e *bad_compare* (linha 8947). Elas são compiladas só se a macro *DEBUG* for definida como *TRUE*. Elas suportam as macros em *assert.h*. Embora não sejam referenciadas em nenhum código discutido neste texto, elas podem ser úteis na depuração para o leitor que quer criar uma versão modificada do MINIX.

2.7 RESUMO

Para ocultar os efeitos das interrupções, os sistemas operacionais fornecem um modelo conceitual que consiste de processos seqüenciais que executam em paralelo. Os processos podem comunicar-se uns com os outros, utilizando primitivas de comunicação interprocesso, como semáforos, monitores ou mensagens. Essas primitivas também são utilizadas para assegurar que nunca dois processos entrem em suas seções críticas ao mesmo tempo. Um processo pode estar executando, executável (pronto) ou bloqueado e pode mudar de estado quando ele ou outro processo executar uma das primitivas de comunicação interprocesso.

As primitivas de comunicação interprocesso podem ser utilizadas para resolver problemas como o dos produtores e consumidores, o dos filósofos jantando, o dos leitores-escritores e o do barbeiro adormecido. Mesmo com essas primitivas, é preciso tomar cuidado para evitar erros e impasses. Muitos algoritmos de agendamento são conhecidos, incluindo *round robin*, agendamento por prioridade, filas de múltiplos níveis e agendadores orientados por política.

O MINIX suporta o conceito de processo e fornece mensagens para a comunicação interprocesso. As mensagens não são armazenadas, portanto, um SEND só tem sucesso quando o destinatário está esperando-o. De maneira semelhante, um RECEIVE só tem sucesso quando uma mensagem já está disponível. Se qualquer uma dessas operações não é bem-sucedida, o processo que fez a chamada é bloqueado.

Quando uma interrupção ocorre, o nível mais baixo do *kernel* cria e envia uma mensagem à tarefa associada com o dispositivo de interrupção. Por exemplo, a tarefa de disco chama *receive* e é bloqueada depois de gravar um comando no hardware controlador de disco requisitando a leitura de um bloco de dados. O hardware do controlador gera uma interrupção quando os dados estão prontos. O software de baixo nível, então, cria uma mensagem para a tarefa de disco e marca-a como executável. Quando o agendador escolhe a tarefa de disco a executar, ela recebe e processa a mensagem. Também é possível que o manipulador de interrupção faça algum trabalho diretamente, como uma interrupção de relógio para atualizar a hora.

A comutação de tarefas pode seguir-se a uma interrupção. Quando um processo é interrompido, uma pilha é criada dentro da entrada do processo na tabela de processos e todas as informações necessárias para reiniciá-la são colocadas na nova pilha. Qualquer processo pode ser reiniciado configurando o ponteiro de pilha para apontar para sua entrada de tabela de processos e iniciando uma seqüência de instruções a fim de restaurar os registradores de CPU, culminando com uma instrução iretd. O agendador decide qual entrada da tabela de processos colocar no ponteiro da pilha.

Interrupções também ocorrem quando o próprio *kernel* está executando. A CPU detecta isso, e a pilha do *kernel*, em vez de uma pilha dentro da tabela de processos, é utilizada. Assim interrupções aninhadas podem ocorrer e quando uma rotina de serviço de interrupção posterior termina, a anterior pode completar-se. Quando todas as interrupções foram servidas, um processo é reiniciado.

O algoritmo de agendamento do MINIX utiliza três filas de prioridade, a mais alta para tarefas, a próxima para o sistema de arquivos, o gerenciador de memória e outros servidores, se houver algum, e a mais baixa para processos de usuário. Os processos de usuário são executados em *round robin* pelo tempo de um quantum por vez. Todos os demais são executados até que bloqueiem ou sofram preempção.

EXERCÍCIOS

1. Suponha que você vá projetar uma arquitetura avançada de computador que faz a comutação de processos em hardware, em vez de ter interrupções. Que informações a CPU precisaria? Descreva como a comutação de processos por hardware pode funcionar.

2. Em todos computadores atuais, pelo menos parte dos manipuladores de interrupções são escritos em linguagem *assembly*. Por quê?

3. No texto, afirmou-se que o modelo da Figura 2-6(a) não se ajustava a um servidor de arquivos utilizando um *cache* em memória. Por que não? Cada processo poderia ter seu próprio *cache*?

4. Em um sistema com *threads*, há uma pilha por *thread* ou uma pilha por processo? Explique.

5. O que é uma condição de corrida?

6. Escreva um *script* de *shell* que produz um arquivo de números seqüenciais lendo o último número no arquivo, adicionando 1 a ele e, então, anexando o resultado ao arquivo. Execute uma instância do *script* em segundo plano e uma em primeiro plano, cada uma acessando o mesmo arquivo. Quanto tempo se passa antes de uma condição de corrida manifestar-se? Qual é a seção crítica? Modifique o *script* para evitar a condição de corrida (sugestão: utilize

   ```
   ln file file.lock
   ```

   para bloquear o arquivo de dados).

7. Uma declaração como

   ```
   ln file file.lock
   ```

   é um mecanismo de bloqueio efetivo para um programa de usuário como os *scripts* utilizados no problema anterior? Por quê (ou por que não)?

8. A solução da espera ativa utilizando a variável *turn* (Figura 2-8) funciona quando os dois processos estão executando em um multiprocessador com memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?

9. Considere um computador que não tem uma instrução TEST AND SET LOCK mas tem uma instrução para comutar conteúdo de um registrador e de uma palavra da memória em uma única ação indivisível. É possível utilizar isso para escrever uma rotina *enter_region* como a encontrada na Figura 2-10?

10. Faça um esboço de como um sistema operacional que pode desativar interrupções poderia implementar semáforos.

11. Mostre como semáforos de contagem (i. e., semáforos que podem armazenar um valor arbitrariamente grande) podem ser implementados utilizando apenas semáforos binários e instruções comuns de máquina.

12. Na Seção 2.2.4, foi descrita uma situação com um processo de alta prioridade, H, e um processo de baixa prioridade, L, que levou a um laço eterno de H. O mesmo problema ocorre se o agendamento por *round robin* for utilizado em vez de agendamento por prioridade? Discuta.

13. A sincronização dentro de monitores utiliza variáveis de condição e duas operações especiais WAIT e SIGNAL. Uma forma mais geral de sincronização seria ter uma única primitiva WAITUNTIL que tivesse um predicado booleano arbitrário como parâmetro. Portanto, poderia dizer-se, por exemplo,

    WAITUNTIL $x < 0$ **or** $y + z < n$

    A primitiva SIGNAL não seria mais necessária. Esse esquema é claramente mais geral que o de Hoare ou Brinch Hansen, mas não é utilizado. Por que não? (Sugestão: pense na implementação.)

14. Um restaurante *fast food* tem quatro tipos de empregados: (1) os atendentes, que anotam os pedidos dos clientes; (2) os cozinheiros, que preparam o alimento; (3) os embaladores, que colocam o alimento em sacolas; (4) os caixas, que entregam as sacolas aos clientes e recebem o dinheiro. Cada empregado pode ser considerado como um processo de comunicação seqüencial. Que forma de comunicação interprocesso eles utilizam? Relacione esse modelo com processos no MINIX.

15. Suponha que temos um sistema de passagem de mensagens utilizando caixas de correio. Ao enviar para uma caixa de correio cheia ou ao tentar receber de uma vazia, um processo não é bloqueado. Em vez disso, ele recebe de volta um código de erro. O processo responde ao código de erro apenas tentando novamente, repetidamente, até ter sucesso. Esse esquema leva a condições de corrida?

16. Na solução para o problema dos filósofos jantando (Figura 2-20), por que a variável de estado é configurada como *HUNGRY* no procedimento *take_forks*?

17. Considere o procedimento *put_forks* na Figura 2-20. Suponha que a variável *state[i]* tenha sido configurada como *THINKING após* as duas chamadas a *test*, em vez de *antes*. Como essa alteração afetaria a solução para o caso de três filósofos? E para 100 filósofos?

18. O problema dos leitores e dos escritores pode ser formulado de várias maneiras com referência a qual categoria de processos pode ser iniciada e quando ela pode ser iniciada. Cuidadosamente, descreva três variações diferentes do problema, cada uma favorecendo (ou não) alguma categoria de processos. Para cada variação, especifique o que acontece quando um leitor ou um escritor torna-se pronto para acessar o banco de dados e o que acontece quando um processo termina de utilizar o banco de dados.

19. Os computadores CDC 6600 podiam gerenciar até 10 processos de E/S simultaneamente utilizando uma forma interessante de agendamento por *round robin* chamada **compartilhamento do processador.** Uma comutação de processos ocorria depois de cada instrução, assim a instrução 1 vinha do processo 1, a instrução 2 vinha do processo, 2 etc. A comutação de processos era feita por hardware especial e o acréscimo (*overhead*) era zero. Se um processo precisasse de *T* s para completar-se na ausência de concorrência, quanto tempo precisaria se o compartilhamento do processador fosse utilizado com *n* processos?

20. Os agendadores por *round robin* normalmente mantêm uma lista de todos os processos executáveis, com cada processo ocorrendo exatamente uma vez na lista. O que aconteceria se um processo ocorresse duas vezes na lista? Você consegue imaginar qualquer razão para permitir isto?

21. Medidas de um certo sistema mostraram que, na média, as execuções dos processos tendiam para um tempo *T* antes de bloquear em E/S. Uma comutação de processos requer um tempo *S*, que é efetivamente desperdiçado (*overhead*). Para agendamento por *round robin* com quantum Q, dê uma fórmula da eficiência de CPU para cada um dos seguintes.
(a) $Q = \alpha$
(B) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) Q perto de 0

22. Cinco *jobs* estão esperando para ser executados. Seus tempos esperados de execução são 9, 6, 3, 5 e *X*. Em que ordem eles devem ser executados para minimizar tempo médio de resposta? (Sua resposta dependerá de *x*.)

23. Cinco *jobs* de lote, *A* até *E*, chegam a um centro de computação quase ao mesmo tempo. Eles têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, com 5 sendo a maior prioridade. Para cada um dos seguintes algoritmos de agendamento, determine o tempo de retorno médio dos processos. Ignore o acréscimo (*overhead*) da comutação de processos.
(a) *Round robin*.
(b) Agendamento por prioridade.
(c) Primeiro a chegar, primeiro a ser servido (execução na ordem 10, 6, 2, 4, 8).
(d) *Job* mais curto primeiro.

Para (a), suponha que o sistema é multiprogramado e que cada *job* receba sua justa parte da CPU. Para (b) a (d) suponha que só um *job* execute por vez, até terminar. Todos os *jobs* são completamente associados à CPU.

24. Um processo que executa no CTSS precisa de 30 quanta para completar-se. Quantas vezes ele deve sofrer comutação, incluindo a primeira vez (antes de ele executar completamente)?

25. O algoritmo de envelhecimento com $a = 1/2$ está sendo utilizado para prever tempos de execução. As quatro execuções anteriores, da mais antiga à mais recente, foram 40, 20, 40 e 15ms. Qual é a previsão do próximo tempo?

26. Um sistema *soft real time* tem quatro eventos periódicos com períodos de 50, 100, 200 e 250ms cada. Suponha que os quatro eventos requeiram 35, 20, 100 e *x* ms de tempo de CPU, respectivamente. Qual é o maior valor de *x* para o qual o sistema é agendável?

27. Explique por que o agendamento de dois níveis é comumente utilizado.

28. Durante execução, o MINIX mantém uma variável *proc_ptr* que aponta para entrada da tabela de processos para o processo atual. Por quê?

29. O MINIX não armazena mensagens. Explique como essa decisão de projeto causa problemas com interrupções de relógio e teclado.

30. Quando uma mensagem é enviada a um processo adormecido no MINIX, o procedimento *ready* é chamado para colocar esse processo na fila adequada de agendamento. Esse procedimento inicia desativando as interrupções. Explique.

31. O procedimento do MINIX *mini_rec* contém um laço. Explique para que ele serve.

32. Essencialmente o MINIX utiliza o método de agendamento na Figura 2-23, com prioridades diferentes para classes. A classe mais baixa (processos de usuário) tem agendamento por *round robin*, mas as tarefas e os servidores sempre têm permissão para executar até bloquearem. É possível que processos na classe mais baixa morram de fome? Por quê (ou por que não)?

33. O MINIX é adequado para aplicativos de tempo real, como registro de dados? Se não, o que poderia ser feito para torná-lo adequado?

34. Suponha que você tenha um sistema operacional que forneça semáforos. Implemente um sistema de mensagens. Escreva os procedimentos para enviar e para receber mensagens.

35. Um aluno forte em Antropologia, mas fraco em Ciência da Computação envolveu-se em um projeto de pesquisa para verificar se babuínos africanos podem ser ensinados sobre impasses. Ele encontrou um desfiladeiro e amarrou uma corda através dele, de modo que os babuínos pudessem cruzá-lo utilizando a corda. Vários babuínos podem cruzar o desfiladeiro ao mesmo tempo, desde que todos sigam na mesma direção. Se os babuínos utilizarem a corda para cruzar o desfiladeiro de leste para oeste e de oeste para leste ao mesmo tempo, o resultado será um impasse (os babuínos ficarão presos no meio) porque é impossível um babuíno

subir sobre outro enquanto suspenso sobre o desfiladeiro. Se um babuíno quiser cruzar o desfiladeiro, ele deve verificar se nenhum outro está atualmente cruzando na direção oposta. Escreva um programa, utilizando semáforos, que evite o impasse. Não se preocupe com uma série de babuínos que se move para leste segurando indefinidamente babuínos que se movem para o oeste.

36. Repita o problema anterior, mas agora evite a fome. Quando um babuíno que quiser cruzar para o leste chegar à corda e encontrar babuínos cruzando para o oeste, ele espera até que a corda esteja vazia, mas os babuínos que se movem para o oeste não mais são permitidos a iniciar até que pelo menos um babuíno tenha cruzado para o outro lado.

37. Resolva o problema dos filósofos jantando utilizando monitores em vez de semáforos.

38. Adicione código ao *kernel* do MINIX para monitorar o número de mensagens enviadas a partir do processo (ou tarefa) *i* ao processo (ou tarefa) *j*. Imprima essa matriz quando a tecla F4 for pressionada.

39. Modifique o agendador do MINIX para monitorar o tempo de CPU que cada processo de usuário teve recentemente. Quando nenhuma tarefa ou servidor quiser executar, selecione o processo de usuário que teve a menor porção da CPU.

40. Reprojete o MINIX de tal modo que cada processo tenha um campo de nível prioridade em sua tabela de processos que possa ser utilizado para dar prioridades mais altas ou mais baixas a processos individuais.

41. Modifique as macros *hwint_master* e *hwint_slave* em *mpx386.s* para que as operações agora realizadas pela função *save* sejam executadas *inline*. Qual é o custo em termos do tamanho do código? Você pode medir um aumento no desempenho?

3

Entrada/Saída

Uma das principais funções de um sistema operacional é controlar todos os dispositivos de E/S (Entrada/Saída). Ele deve enviar comandos para os dispositivos, capturar interrupções e tratar erros. Também deve oferecer uma interface entre os dispositivos e o restante do sistema que seja simples e fácil de usar. Na medida do possível, a interface deve ser a mesma para todos os dispositivos (independência de dispositivo). O código de E/S representa uma fração significativa do total do sistema operacional.

A maneira como o sistema operacional gerencia E/S é o assunto deste capítulo, cuja visão geral é apresentada a seguir.

Primeiro veremos um resumo sobre alguns princípios de hardware de E/S e, então, veremos software de E/S que, em geral, pode ser estruturado em camadas, com cada camada tendo uma tarefa bem definida a executar. Estudaremos essas camadas para ver o que elas fazem e como seu conjunto é organizado.

Logo após, vem uma seção sobre impasses (*deadlocks*). Definiremos impasses precisamente, mostrando como são causados, oferecendo dois modelos para analisá-los e discutindo alguns algoritmos para prevenir sua ocorrência. Então, daremos uma rápida olhada em E/S no MINIX. Seguindo-se a essa introdução, veremos quatro dispositivos de E/S em detalhes — o disco de RAM, o disco rígido, o relógio e o terminal. Para cada dispositivo, estudaremos seu hardware, seu software e implementação no MINIX. Por fim, o capítulo fecha com uma breve discussão sobre uma pequena parte do MINIX que está localizada na mesma camada que as tarefas de E/S, mas que não é propriamente uma tarefa de E/S. Ela oferece alguns serviços para o gerenciador de memória e para o sistema de arquivos, como buscar blocos de dados de um processo de usuário.

3.1 PRINCÍPIOS DE HARDWARE DE E/S

Pessoas diferentes vêem o hardware de E/S de maneiras diferentes. Engenheiros elétricos vêem-no em termos de chips, de fios, de fontes de alimentação, de motores e de todos os outros componentes físicos que o constituem. Os programadores vêem a interface apresentada para o software — os comandos que o hardware aceita, as funções que ele executa e os erros que podem ser retornados. Neste livro, estamos preocupados com a programação de dispositivos de E/S, não como projetar, como construir ou como mantê-los, portanto nosso interesse irá restringir-se ao modo como o hardware é programado, não como ele funciona por dentro. Contudo, a programação de dispositivos de E/S com freqüência está intimamente associada com sua operação interna. Nas próximas três seções, ofereceremos uma breve fundamentação sobre hardware de E/S no que diz respeito à programação.

3.1.1 Dispositivos de E/S

Os dispositivos de E/S podem ser divididos, grosso modo, em duas categorias: **dispositivos de bloco** e **dispositivos de caractere.** Um dispositivo de bloco armazena informações em blocos de tamanho fixo, cada um com seu próprio endereço. Os tamanhos de bloco comuns variam de 512 a 32.768 bytes. A propriedade essencial de um dispositivo de bloco é que é possível ler ou gravar cada bloco independentemente de todos os outros. Os discos são os dispositivos de bloco mais comuns.

Se você olhar de perto, verá que não é bem definida a divisão entre dispositivos que são endereçáveis por blocos e aqueles que não são. Todo o mundo concorda que um

disco é um dispositivo endereçável por blocos, porque, independentemente de onde o braço está atualmente, sempre é possível buscar outro cilindro e, então, esperar o bloco requisitado passar sob o cabeçote. Agora considere uma unidade de fita DAT ou de 8 mm utilizada para fazer backups de disco. Suas fitas geralmente contêm blocos de tamanho fixo. Se a unidade de fita receber um comando para ler o bloco *N*, ela pode sempre retroceder e avançar a fita até chegar ao bloco *N*. Essa operação é análoga a um disco fazendo uma busca, exceto que toma muito mais tempo. Além disso, pode ou não ser possível regravar um bloco no meio de uma fita. Mesmo que fosse possível utilizar fitas como dispositivos de bloco de acesso aleatório, isso ultrapassaria um tanto o limite estabelecido: elas normalmente não são utilizadas assim.

O outro tipo de dispositivo de E/S é o dispositivo de caractere, o qual entrega ou aceita um fluxo de caracteres, sem considerar qualquer estrutura de bloco. Ele não é endereçável e tampouco tem qualquer operação de busca. As impressoras, interfaces de rede, mouses (para apontar), ratos (para experiências de laboratório de Psicologia) e a maioria dos outros dispositivos que não são do tipo disco podem ser vistas como dispositivos de caractere.

Esse esquema de classificação não é perfeito. Alguns dispositivos simplesmente não se ajustam nele. Os relógios, por exemplo, não são endereçáveis por bloco. Tampouco eles geram ou aceitam fluxos de caractere. Tudo que fazem é gerar interrupções em intervalos bem definidos. As telas mapeadas em memória também não se ajustam ao modelo. De qualquer modo, o modelo de dispositivos de bloco e de caractere é suficientemente geral para que possa ser utilizado como uma base para construir, de forma independente de dispositivo, algumas partes do sistema operacional que tratam de E/S. O sistema de arquivos, por exemplo, lida apenas com dispositivos de bloco abstratos e deixa a parte dependente do dispositivo para o software de baixo nível chamado ***drivers* de dispositivo.**

3.1.2 Controladoras de Dispositivo

As unidades de E/S geralmente consistem em um componente mecânico e em outro eletrônico. Freqüentemente é possível separar as duas partes para oferecer um projeto mais modular e genérico. O componente eletrônico é chamado **controladora** ou **adaptador de dispositivo**. Em computadores pessoais, esse freqüentemente toma a forma de uma placa de circuito impresso que pode ser inserida em um *slot* na ***parentboard*** do computador (antes incorretamente chamada de ***motherboard***, placa-mãe). O componente mecânico é o dispositivo em si.

A placa controladora normalmente tem nela um conector, onde um cabo que leva ao dispositivo em si pode ser conectado. Muitas controladoras podem manipular, quatro ou até oito dispositivos idênticos. Se a interface entre a controladora e o dispositivo é uma interface-padrão, seja um dos padrões oficiais como ANSI, IEEE ou ISO ou um padrão *de fato*, então as empresas podem fazer controladoras ou dispositivos que se ajustam a essa interface. Muitas empresas, por exemplo, fazem unidades de disco que se ajustam aos padrões de interfaces controladoras de disco IDE (*Integrated Drive Electronics*) ou SCSI (*Small Computer System Interface*).

Mencionamos essa distinção entre controladora e dispositivo porque o sistema operacional quase sempre lida com a controladora, não com o dispositivo. A maioria dos pequenos computadores utiliza o modelo de barramento único da Figura de 3-1 para comunicação entre a CPU e as controladoras. *Mainframes* freqüentemente utilizam um modelo diferente, com múltiplos barramentos e computadores especializados de E/S, chamados **canais de E/S** que assumem parte da carga da CPU principal.

A interface entre a controladora e o dispositivo é freqüentemente uma interface de muito baixo nível. Um disco, por exemplo, talvez seja formatado com 16 setores de 512 bytes por trilha. Entretanto, o que realmente sai da unidade é um fluxo serial de bits, iniciando com um **preâmbulo,** depois os 4096 bits em um setor e por fim uma soma de verificação, também chamada **Código para Correção de Erros — ECC *(Error-Correcting Code)***. O preâmbulo é gravado quando o disco é formatado e contém o cilindro e o número de setor, o tamanho do setor e dados semelhantes, assim como as informações de sincronização.

O trabalho da controladora é converter o fluxo serial de bits em um bloco de bytes e executar qualquer correção de erro necessária. O bloco de bytes tipicamente é primeiro

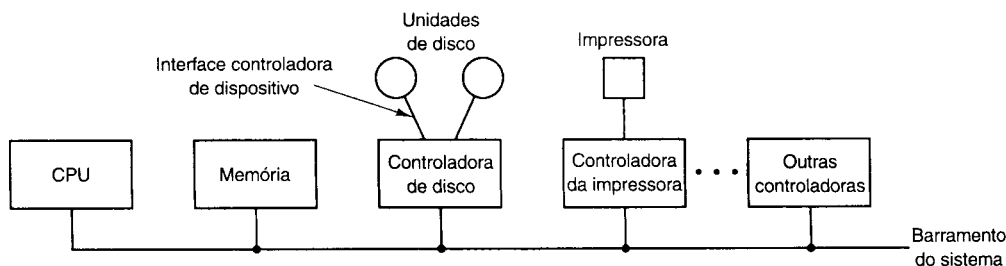



Figura 3-1 Um modelo para conectar CPU, memória, controladoras e dispositivos de E/S.

montado, bit por bit, em um buffer dentro da controladora. Depois que sua soma de verificação foi verificada e o bloco foi declarado livre de erros, ele pode, então, ser copiado para a memória principal.

A controladora para um terminal CRT também funciona como um dispositivo serial de bits em um nível igualmente baixo. Ela lê da memória bytes que contêm os caracteres a serem exibidos e gera os sinais utilizados para modular o feixe do CRT a fim de fazê-lo escrever na tela. A controladora também gera os sinais para instruir o feixe do CRT a fazer um retraço horizontal depois de ele acabar uma linha de varredura, bem como os sinais para instruí-lo a fazer um retraço vertical depois que a tela inteira foi varrida. Se isso não fosse feito pela controladora do CRT, o programador do sistema operacional teria de programar explicitamente uma varredura análoga para o tubo. Com a controladora, o sistema operacional inicia a controladora com alguns parâmetros, como o número de caracteres por linha e número de linhas por tela e deixa a controladora encarregada de realmente guiar o feixe.

Cada controladora tem alguns registradores que são utilizados para comunicar-se com a CPU. Em alguns computadores, esses registradores são parte do espaço normal de endereçamento de memória. Esse esquema é chamado **E/S mapeada em memória.** O 680x0, por exemplo, utiliza esse método. Outros computadores utilizam um espaço especial de endereçamento para E/S, com cada controladora alocando uma certa parte dele. A atribuição de endereços de E/S para dispositivos é feita pela lógica de decodificação de barramento associada com a controladora. Alguns fabricantes dos chamados compatíveis com IBM PC utilizam endereços diferentes de E/S diferentes daqueles usados pela IBM. Além de portas de E/S, muitas controladoras utilizam interrupções para informar à CPU quando estão prontas para ter seus registradores lidos ou gravados. Uma interrupção é, em primeiro lugar, um evento elétrico. Uma linha de solicitação de interrupção de hardware (IRQ — *Interrupt ReQuest*) é uma entrada física para o chip controlador de interrupções. O número dessas entradas é limitado; PCs da classe Pentium têm somente 15 disponíveis para dispositivos de E/S. Algumas controladoras são conectadas diretamente (*hard-wired*) à placa-mãe do sistema, como é, por exemplo, a controladora de teclado de um IBM PC. No caso de uma controladora que se conecta ao *backplane*,* às vezes, podem ser utilizados *switches* ou *jumpers* na controladora de dispositivo para selecionar a IRQ que o dispositivo utilizará, a fim de evitar conflitos (embora em algumas placas, como as *Plug and Play*, as IRQs possam ser configuradas por software). O chip controlador de interrupções mapeia cada entrada de IRQ para um vetor de interrupção, que localiza o correspondente software de serviço de interrupção. A Figura 3-2 mostra os endereços de E/S, as interrupções de hardware e o vetor de interrupção atribuídos a algumas controladoras em um IBM PC, a título de exemplo. O MINIX utiliza as mesmas interrupções de hardware, mas o vetores de interrupção do MINIX são diferentes desses mostrados aqui para MS-DOS.

O sistema operacional executa E/S gravando comandos nos registradores da controladora. A controladora de disquetes do IBM PC, por exemplo, aceita 15 comandos diferentes, como READ, WRITE, SEEK, FORMAT e RECALIBRATE. Muitos dos comandos têm parâmetros, que também são carregados nos registradores da controladora. Quando um comando é aceito, a CPU pode deixar a controladora continuar sozinha e ir fazer outro trabalho. Quando o comando é completado, a controladora gera uma interrupção para permitir que o sistema operacional ganhe controle da CPU e teste os resultados da operação. A CPU obtém os resultados e o status do dispositivo lendo um ou mais bytes de informações dos registradores da controladora.

3.1.3 Acesso Direto à Memória (DMA)

Muitas controladoras, especialmente as de dispositivos de bloco, suportam **Acesso Direto à Memória ou DMA (*Direct Memory Access*)**. Para explicar como o DMA funciona vejamos primeiro como as leituras de disco ocorrem quando o DMA não é utilizado. Primeiro a controladora lê o bloco (um ou mais setores) da unidade serialmente, bit a bit, até que o bloco inteiro esteja no buffer interno da controladora. Em seguida, ela calcula a soma de verificação para certificar-se de que não ocorreram erros de leitura. Então, a controladora gera uma interrupção. Quando o sistema operacional começa a executar, ele pode ler o bloco de disco do buffer da controladora, um byte ou uma palavra por vez executando um laço, com cada iteração lendo um byte ou uma palavra de um registrador da controladora de dispositivo e armazenando-o na memória.

Naturalmente, um laço programado para ler os bytes um por vez a partir da controladora desperdiça tempo da CPU. O DMA foi inventado para liberar a CPU desse trabalho de baixo nível. Quando é utilizado, a CPU fornece dois itens de informação para a controladora, além do endereço do bloco no disco: o endereço de memória para onde o bloco deve ir e o número de bytes a transferir, como mostrado na Figura 3-3.

Depois que a controladora leu o bloco inteiro do dispositivo para seu buffer e verificou a soma de verificação, ela copia o primeiro byte ou palavra para o endereço na memória principal especificado pelo endereço de memória DMA. Então, ela incrementa o endereço de DMA e decrementa a contagem do DMA pelo número de bytes que acabou de transferir. Esse processo é repetido até que a contagem de DMA torne-se zero, momento em que a controladora gera uma interrupção. Quando o sistema operacional inicia, ele não precisa copiar o bloco para a memória; ele já está lá.

*N. de T. Uma placa ou uma estrutura de circuitos que suporta outras placas de circuitos, dispositivos e as interconexões entre os dispositivos, e fornece energia e sinais de dados aos dispositivos suportados. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

Controladora de E/S	Endereço de E/S	IRQ de hardware	Vetor de interrupção
Relógio	040 - 043	0	8
Teclado	060 - 063	1	9
Disco rígido	1F0 - 1F7	14	118
RS232 secundário	2F8 - 2FF	3	11
Impressora	378 - 37F	7	15
Disquete	3F0 - 3F7	6	14
RS232 primário	3F8 - 3FF	4	12

Figura 3-2 Alguns exemplos de controladoras, seus endereços de E/S, suas linhas de interrupção de hardware e seu vetor de interrupção em um PC típico rodando MS-DOS.

Você pode estar perguntando-se por que a controladora simplesmente não armazena os bytes na memória principal logo que os recebe do disco. Em outras palavras, por que ela precisa de um buffer interno? A razão é que uma vez que a transferência de disco iniciou, os bits continuam chegando do disco a uma taxa constante, esteja a controladora pronta para eles ou não. Se a controladora tentasse gravar dados diretamente na memória, ela teria de passar pelo barramento de sistema para cada palavra transferida. Se o barramento estivesse ocupado devido a algum outro dispositivo que o estivesse utilizando, a controladora teria de esperar. Se a próxima palavra do disco chegasse antes que a anterior tivesse sido armazenada, a controladora teria de armazená-la em algum lugar. Se o barramento estivesse muito ocupado, a controladora talvez acabasse armazenando muitas palavras e teria muita administração a fazer também. Quando o bloco é *bufferizado* internamente, o barramento não é necessário até que o DMA inicie; então, o projeto da controladora é muito mais simples porque a transferência via DMA para a memória não é uma tarefa dependente do tempo. (Algumas controladoras mais velhas iam, de fato, diretamente para memória com apenas um pequeno buffer interno, mas quando o barramento estava muito ocupado, uma transferência poderia precisar ser terminada com um erro de *overrun*.*)

O processo de *bufferização* em duas etapas descrito acima tem implicações importantes para o desempenho de E/S. Enquanto os dados estão sendo transferidos da controladora para a memória, seja pela CPU ou pela controladora, o próximo setor estará passando sob o cabeçote do disco, os bits estarão chegando na controladora. Controladoras mais simples não conseguem fazer presente a entrada e a saída simultâneas, então, enquanto uma transferência de memória está acontecendo, o setor que passa sob o cabeçote do disco é perdido.

Como resultado, a controladora somente será capaz de ler blocos alternados. A leitura de uma trilha completa, então, exigirá duas rotações completas, uma para os blocos pares e uma para os blocos ímpares. Se o tempo de transferir um bloco da controladora para a memória pelo barramento for mais longo que o tempo de ler um bloco do disco, pode ser necessário ler um bloco e, então, pular dois (ou mais) blocos.

A técnica de saltar blocos a fim de dar tempo para a controladora transferir dados para a memória é chamada **intercalação** (*interleaving*). Quando o disco é formatado, os blocos são numerados para tomar conta do fator de intercalação. Na Figura 3-4(a) vemos um disco com 8 blocos por trilha e nenhuma intercalação. Na Figura 3-4(b) vemos o mesmo disco com uma única intercalação. Na Figura 3-4(c), a intercalação dupla é mostrada.

A idéia de numerar os blocos dessa maneira é para permitir que o sistema operacional leia os blocos consecutivamente numerados e ainda alcance a taxa máxima de que o hardware é capaz. Se os blocos foram numerados, como na Figura 3-4(a), mas a controladora apenas consegue ler blocos alternados, um sistema operacional que alocasse um arquivo de oito blocos em blocos de disco consecutivos exigiria oito rotações de disco para ler os blocos de 0 a 7 em ordem. (Naturalmente, se o sistema operacional soubesse do problema e alocasse seus blocos de maneira diferente, ele poderia resolver o problema no nível de software, mas é melhor ter a controladora preocupando-se com a intercalação.)

Nem todos os computadores utilizam DMA. O argumento contra é que a CPU principal é freqüentemente muito mais rápida que a controladora de DMA e pode fazer o trabalho muito mais rápido (quando o fator limitante não é a velocidade do dispositivo de E/S). Se não houver outro trabalho para ela fazer, ter a (rápida) CPU esperando a (lenta) controladora de DMA é algo sem sentido. Além disso, livrar-se da controladora de DMA e ter a CPU fazendo todo o trabalho em software economiza algum dinheiro.

*N. de T. Na transferência de informações, um erro que ocorre quando um dispositivo receptor não consegue tratar ou utilizar os dados com a mesma rapidez com que são enviados. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

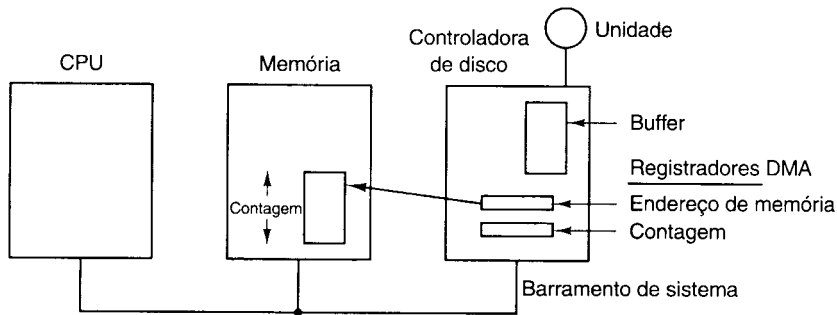



Figura 3-3 Uma transferência de DMA é feita inteiramente pela controladora.

3.2 PRINCÍPIOS BÁSICOS DO SOFTWARE DE E/S

Permita-nos desviar do hardware por um momento e ver como o software de E/S é estruturado. As metas gerais do software de E/S são fáceis de declarar. A idéia básica é organizar o software como uma série de camadas, com as mais baixas preocupadas em esconder as peculiaridades do hardware das mais altas e estas últimas preocupadas em apresentar uma interface amigável, limpa e simples aos usuários. Nas seções a seguir veremos essas metas e como elas são alcançadas.

3.2.1 Metas do Software de E/S

Um conceito-chave no projeto de software de E/S é conhecido como **independência de dispositivo**. Isso significa que deve ser possível escrever programas que podem ler arquivos em um disquete, em um disco rígido ou em um CD-ROM, sem que seja necessário modificar os programas para cada tipo de dispositivo diferente. Qualquer um deve ser capaz de digitar um comando como

```
sort <input> output
```

e fazê-lo funcionar com a entrada proveniente de um disquete de um disco rígido ou o teclado e a saída indo para o disquete, para o disco rígido ou até para a tela. Cabe ao sistema operacional cuidar dos problemas causados pelo fato de que esses dispositivos realmente são diferentes e requerem *drivers* de dispositivo muito diferentes para realmente gravar os dados no dispositivo de saída.

Intimamente relacionada com a independência de dispositivo está a meta de **atribuição uniforme de nomes.** O nome de um arquivo ou de um dispositivo deve ser simplesmente uma *string* ou um número inteiro e não depender do dispositivo de nenhuma maneira. No UNIX, todos os discos podem estar integrados juntos na hierarquia do sistema de arquivos de maneiras arbitrárias para que o usuário não precise saber qual nome corresponde a qual dispositivo. Por exemplo, um disquete pode ser **montado** no diretório */usr/ast/backup* de tal modo que a ação de copiar um arquivo para */usr/ast/backup/monday* copie o arquivo para o disquete. Assim, todos os arquivos e os dispositivos são endereçados da mesma maneira: por um nome de caminho.

Outra questão importante para o software de E/S é o tratamento de erros. Em geral, os erros devem ser tratados o mais perto possível do hardware. Se a controladora descobrir um erro de leitura, ela deverá tentar corrigir o erro se puder. Se não puder, então o *driver* de dispositivo deverá tratá-lo, talvez tentando simplesmente ler o bloco novamente. Muitos erros são transitórios, como aqueles de lei-

(a)

(b)

(c)

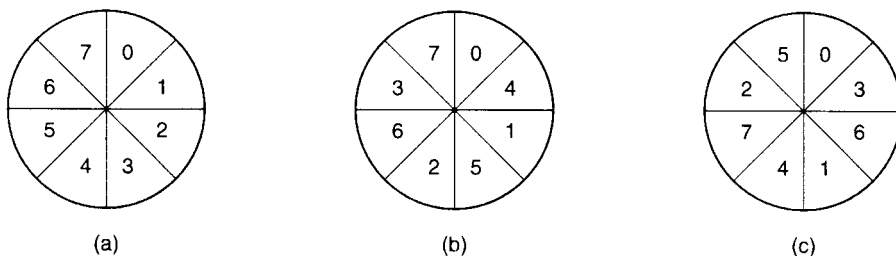



Figura 3-4 (a) Nenhuma intercalação. (b) Intercalação única. (c) Intercalação dupla.

tura causados por partículas de pó no cabeçote de leitura e desaparecem se a operação é repetida. Somente quando as camadas mais baixas não são capazes de lidar com o problema é que as camadas superiores devem ser informadas. Em muitos casos, a recuperação de erros pode ser feita transparentemente em um nível baixo, sem que os níveis superiores nem mesmo saibam sobre o erro.

Outra questão-chave são as transferências síncronas (com bloqueio) *versus* assíncronas (baseadas em interrupções). A maior parte da E/S é assíncrona — a CPU inicia a transferência e segue adiante para fazer outra coisa até a interrupção chegar. Os programas de usuário são muito mais fáceis de escrever se as operações de E/S provocarem bloqueios — depois de um comando READ o programa é automaticamente suspenso até que os dados estejam disponíveis no buffer. Cabe ao sistema operacional fazer com que as operações que são de fato baseadas em interrupções parecerem-se com bloqueios para os programas do usuário.

O conceito final com que lidaremos aqui é dispositivos compartilháveis *versus* dispositivos dedicados. Alguns dispositivos de E/S, como discos, podem ser utilizados por muitos usuários ao mesmo tempo. Nenhum problema é causado por múltiplos usuários tendo arquivos abertos no mesmo disco ao mesmo tempo. Outros dispositivos, como unidades de fita, precisam ser dedicados a um único usuário até que esse usuário tenha terminado. Então, outro usuário pode ter a unidade de fita. Ter dois ou mais usuários gravando blocos misturados aleatoriamente na mesma fita definitivamente não funcionará. A introdução de dispositivos dedicados (não-compartilhados) também introduz uma variedade de problemas. Novamente, o sistema operacional deve ser capaz de tratar dispositivos tanto compartilhados como dedicados de uma maneira que evite problemas.

Essas metas podem ser alcançadas de uma maneira eficiente e abrangente estruturando o software de E/S em quatro camadas:

1. Manipuladores de interrupções (fundo).
2. *Drivers* de dispositivo.
3. Software de sistema operacional independente de dispositivo.
4. Software de nível de usuário (topo).

Essas quatro camadas são (não acidentalmente) as mesmas quatro camadas que vimos na Figura 2-26. Nas seções a seguir veremos uma por vez, começando pelo fundo. A ênfase neste capítulo está nos *drivers* de dispositivo (camada 2), mas resumiremos o restante do software de E/S para mostrar como as várias partes do sistema de E/S ajustam-se entre si.

3.2.2 Manipuladores de Interrupções

Interrupções são uma realidade desagradável. Elas devem ser escondidas longe, no fundo das entranhas do sistema operacional, de modo que o mínimo possível do sistema saiba sobre elas. A melhor maneira de ocultá-las é ter cada processo que inicia uma operação de E/S bloqueado até que a E/S tenha-se completado e a interrupção tenha ocorrido. O processo pode bloquear-se fazendo um DOWN em um semáforo, um WAIT em uma variável de condição ou um RECEIVE em uma mensagem, por exemplo.

Quando as interrupções acontecem, o procedimento de interrupção faz o que tem de fazer para desbloquear o processo que iniciou a E/S. Em alguns sistemas, ele fará um UP em um semáforo. Em outros, ele fará um SIGNAL em uma variável de condição em um monitor. Em outros, ainda, ele enviará uma mensagem para o processo bloqueado. Em todos casos, o efeito geral da interrupção será que um processo que anteriormente estava bloqueado agora será capaz de executar.

3.2.3 *Drivers* de Dispositivo

Todo código dependente de dispositivo deve estar nos *drivers* de dispositivos. Cada *driver* de dispositivo trata de um tipo de dispositivo ou, no máximo, de uma classe de dispositivos intimamente relacionados. Por exemplo, provavelmente seria uma boa idéia ter um único *driver* de terminal, mesmo que o sistema suportasse diversos tipos de terminal, todos ligeiramente diferentes. Por outro lado, um terminal burro para a impressão de listagens e um terminal gráfico inteligente com um mouse são tão diferenciados que *drivers* diferentes deverão ser utilizados.

Anteriormente neste capítulo vimos o que as controladoras de dispositivo fazem. Vimos que cada controladora tem um ou mais registradores de dispositivo utilizados para receber comandos. Os *drivers* de dispositivo enviam esses comandos e verificam se eles foram executados adequadamente. Assim, o *driver* de disco é a única parte do sistema operacional que sabe quantos registradores tal controladora de disco tem e para o que eles são utilizados. Sozinho, ele sabe tudo sobre setores, trilhas, cilindros, cabeçotes, movimento do braço, fatores de intercalação, unidades de motor, tempos de acomodação do cabeçote, e todos os outros fatores mecânicos envolvidos no trabalho de fazer o disco funcionar adequadamente.

Em termos gerais, o trabalho de um *driver* de dispositivo é aceitar solicitações abstratas do software independente de dispositivo acima dele e cuidar para que a solicitação seja executada. Uma solicitação típica é ler o bloco *n*. Se o *driver* estiver desocupado no momento em que uma solicitação chega, ele começa a executar a solicitação imediatamente. Se, entretanto, ele já estiver ocupado com uma solicitação, normalmente ele colocará a nova solicitação em uma fila de solicitações pendentes a serem tratadas logo que possível.

O primeiro passo para realmente executar uma solicitação de E/S, digamos, para um disco, é traduzi-la de um termo abstrato para um termo concreto. Para um *driver* de disco, isso significa descobrir onde no disco o bloco requerido realmente está, verificar se o motor da unidade está ligado, determinar se o braço está posicionado no cilindro adequado e assim por diante. Em resumo, ele deve decidir

que operações da controladora são requeridas e em que seqüência.

Uma vez determinados quais comandos serão enviados para a controladora, ele começa a emitir esses comandos gravando nos registradores de dispositivo da controladora. Algumas controladoras podem tratar somente um comando por vez. Outras aceitam uma lista encadeada de comandos, que, então, executam sozinhas, sem ajuda do sistema operacional.

Depois que o comando ou os comandos foram dados, uma de duas situações será aplicada. Em muitos casos, o *driver* de dispositivo deve esperar até que a controladora faça algum trabalho para ele; então, ele bloqueia a si próprio até que as interrupções entrem para desbloqueá-lo. Em outros casos, entretanto, a operação acaba sem demora, assim o *driver* não precisa bloquear. Como um exemplo desta última situação, a ação de rolar a tela em alguns terminais requer simplesmente gravar alguns bytes nos registradores da controladora. Nenhum movimento mecânico é necessário; então, a operação inteira pode ser completada em alguns microssegundos.

No primeiro caso, o *driver* suspenso será acordado pela interrupção. No último caso, ele nunca irá dormir. De qualquer maneira, depois que a operação foi completada, ele deve fazer uma verificação de erros. Se tudo estiver certo, o *driver* pode ter dados para passar para o software independente de dispositivo (p. ex., um bloco recém-lido). Por fim, ele retorna algumas informações de status para informe de erro para quem o chamou. Se quaisquer outras solicitações estiverem enfileiradas, uma delas agora pode ser selecionada e iniciada. Se nada estiver enfileirado, o *driver* é bloqueado e fica aguardando a próxima solicitação.

3.2.4 Software de E/S Independente de Dispositivo

Embora parte do software de E/S seja específico de dispositivo, uma grande parte dele é independente de dispositivo. A divisão exata entre os *drivers* e o software independente de dispositivo depende do sistema, porque algumas funções que poderiam ser feitas de uma maneira independente de dispositivo podem, na realidade, ser feitas nos *drivers*, por eficiência ou por outras razões. As funções mostradas na Figura 3-5 geralmente são feitas no software independente de dispositivo. No MINIX, a maioria do software independente de dispositivo é parte do sistema de arquivos, na camada 3 (Figura 2-26). Embora estudaremos o sistema de arquivos no Capítulo 5, veremos aqui rapidamente o software independente de dispositivo para oferecer uma perspectiva da E/S e mostrar melhor onde os *drivers* enquadram-se.

A função básica do software independente de dispositivo é executar as funções de E/S que são comuns para todos dispositivos e oferecer uma interface uniforme para o software de nível de usuário.

Uma questão importante em um sistema operacional é a maneira como são nomeados objetos como arquivos e dispositivos de E/S. O software independente de dispositivo cuida de mapear nomes simbólicos de dispositivo para o *driver* adequado. No UNIX, um nome de dispositivo, como */dev/tty00*, especifica unicamente o nó-i para um arquivo especial, e esse nó-i contém o **número principal do dispositivo,** que é utilizado para localizar o *driver* apropriado. O nó-i também contém o **número secundário do dispositivo,** que é passado como um parâmetro para o *driver* para especificar a unidade a ler ou a gravar.

Intimamente relacionado com a nomeação está a proteção. Como o sistema impede que os usuários acessem dispositivos aos quais eles não têm direitos de acesso? Na maioria dos sistemas de computador pessoal, não há nenhuma proteção. Qualquer processo pode fazer qualquer coisa que quiser. Na maioria dos sistemas de *mainframe*, o acesso a dispositivos de E/S por processos de usuário é completamente proibido. No UNIX, um esquema mais flexível é utilizado. Os arquivos especiais correspondentes aos dispositivos de E/S são protegidos pelos bits *rwx* normais. O administrador de sistema pode, então, configurar as permissões adequadas para cada dispositivo.

Discos diferentes podem ter tamanhos de setor diferentes. Cabe ao software independente de dispositivo esconder esse fato e oferecer um tamanho uniforme de bloco para as camadas mais altas, por exemplo, tratando vários setores

Interfaceamento uniforme para *drivers* de dispositivo
Nomeação de dispositivo
Proteção de dispositivo
Fornecimento de um tamanho de bloco independente de dispositivo
Bufferização
Alocação de armazenamento em dispositivos de bloco
Atribuição e liberação de dispositivos dedicados
Informe de erros

Figura 3-5 As funções do software de E/S independente de dispositivo.

como um único bloco lógico. Dessa maneira, as camadas mais altas lidam somente com dispositivos abstratos, que utilizam o mesmo tamanho de bloco lógico, independente do tamanho físico do setor. De maneira semelhante, alguns dispositivos de caractere entregam seus dados à freqüência de um byte por vez (p. ex., modems), enquanto outros entregam seus dados em unidades maiores (p. ex., interfaces de rede). Essas diferenças também devem ser ocultadas.

A *bufferização* também é uma questão, tanto para dispositivos de bloco como para os de caractere. Para dispositivos de bloco, o hardware geralmente insiste em ler e em gravar blocos inteiros de uma vez, mas processos de usuário são livres para ler e para gravar em unidades arbitrárias. Se um processo de usuário gravar metade de um bloco, o sistema operacional normalmente manterá os dados internamente até que o resto dos dados sejam gravados, momento em que o bloco pode sair para o disco. Para dispositivos de caractere, os usuários podem gravar dados no sistema mais rapidamente do que ele pode dar saída, precisando, então, de *bufferização*. A entrada de teclado que chega antes de ser necessária também requer *bufferização*.

Quando um arquivo é criado e é preenchido com dados, novos blocos de disco precisam ser alocados para o arquivo. Para executar essa alocação, o sistema operacional precisa de uma lista ou um mapa de bits dos blocos livres por disco, mas o algoritmo para localizar um bloco livre é independente de dispositivo e pode ser feito acima do nível do *driver*.

Alguns dispositivos, como gravadores de CD-ROM, podem ser utilizados somente por um único processo em um momento qualquer. Cabe ao sistema operacional examinar as solicitações para utilização de dispositivo e aceitá-las ou rejeitá-las, dependendo se o dispositivo requerido estiver disponível ou não. Uma maneira simples de tratar essas solicitações é requerer que os processos executem OPEN diretamente nos arquivos especiais para dispositivos. Se o dispositivo estiver indisponível, o OPEN falhará. O fechamento de um dispositivo dedicado iria liberá-lo.

O tratamento de erros, de modo geral, é feito pelos *drivers*. A maioria dos erros é altamente dependente do dispositivo; então, somente o *driver* sabe o que fazer (p. ex., tentar novamente, ignorar, pane). Um erro típico é causado por um bloco de disco danificado que não pode ser mais lido. Depois que o *driver* tentou ler o bloco um certo número de vezes, ele desiste e informa ao software independente de dispositivo. A maneira como o erro é tratado daqui é independente de dispositivo. Se o erro ocorreu durante a leitura de um arquivo de usuário, pode ser suficiente informar o erro ao processo que registrou a leitura. Entretanto, se ele ocorreu durante a leitura de uma estrutura de dados crítica do sistema, como o bloco que contém o mapa de bits mostrando quais blocos estão livres, o sistema operacional não pode ter outra escolha senão imprimir uma mensagem de erro e terminar.

3.2.5 Software de E/S no Espaço do Usuário

Embora a maioria do software de E/S esteja dentro do sistema operacional, uma pequena parte consiste em bibliotecas vinculadas em programas de usuário, e até mesmo programas inteiros que executam fora do *kernel*. As chamadas de sistema, incluindo as chamadas de sistema de E/S, normalmente são feitas por procedimentos de biblioteca. Quando um programa de C contém a chamada

```
count = write(fd, buffer, nbytes);
```

o procedimento de biblioteca *write* será vinculado com o programa e estará contido no programa binário presente na memória em tempo de execução. A coleção de todos esses procedimentos de biblioteca é claramente parte do sistema de E/S.

Embora esses procedimentos façam pouco mais que colocar seus parâmetros no lugar apropriado para a chamada de sistema, há outros procedimentos de E/S que de fato fazem trabalho real. Em particular, a formatação de entrada e de saída é feita pelos procedimentos de biblioteca. Um exemplo de C é *printf*, que pega uma *string* de formato e possivelmente algumas variáveis como entrada, constrói uma *string* de ASCII e, então, chama WRITE para dar saída à *string*. Um exemplo de um procedimento semelhante para entrada é *scanf* que lê a entrada e armazena-a em variáveis descritas em uma *string* de formato que usa a mesma sintaxe de *printf*. A biblioteca-padrão de E/S contém diversos procedimentos que envolvem E/S e todas executam como parte de programas de usuário.

Nem todo software de E/S no nível de usuário consiste em procedimentos de biblioteca. Outra categoria importante é o sistema de *spool*. Fazer *spool* é uma maneira de lidar com dispositivos dedicados de E/S em um sistema de multiprogramação. Considere um dispositivo típico para o qual se faz *spool*: uma impressora. Embora pudesse ser uma técnica fácil deixar qualquer processo de usuário abrir o arquivo especial de caractere para a impressora, suponha que um processo abrisse-o e, então, não fizesse nada durante horas. Nenhum outro processo poderia imprimir qualquer coisa.

Em vez disso, o que é feito é criar um processo especial, chamado ***daemon***, e um diretório especial, chamado **diretório de *spool***. Para imprimir um arquivo, um processo primeiro gera o arquivo inteiro a ser impresso e coloca-o no diretório de *spool*. Cabe ao *daemon*, o único processo a ter permissão para utilizar o arquivo especial da impressora, imprimir os arquivos no diretório. Protegendo o arquivo especial contra o uso direto dos usuários, o problema de ter alguém mantendo-o aberto desnecessariamente é eliminado.

A técnica de *spool* não é utilizada apenas para impressoras. Ela também é utilizada em outras situações, como,

por exemplo, transferência de arquivos por uma rede, quando freqüentemente é utilizado um *daemon* de rede. Para enviar um arquivo para algum lugar, um usuário coloca-o em um diretório de *spool* da rede. Mais tarde, o *daemon* de rede pega-o e transmite-o. Um exemplo particular de transmissão de arquivo utilizando o *spool* é o sistema de correio eletrônico da Internet. Essa rede consiste em milhões de máquinas ao redor do mundo utilizando muita redes de computador que se comunicam entre si. Para enviar uma mensagem para alguém, você chama um programa como *send*, que recebe a carta a ser enviada e, então, deposita-a em um diretório de *spool* para ser transmitida posteriormente. O sistema inteiro de correio executa fora do sistema operacional.

A Figura 3-6 resume o sistema de E/S, mostrando todas as camadas e as principais funções de cada uma. Começando do fundo, as camadas são o hardware, os manipuladores de interrupções, os *drivers* de dispositivo, o software independente de dispositivo e, por fim, os processos de usuário.

As setas na Figura 3-6 mostram o fluxo de controle. Quando um programa de usuário tenta ler um bloco de um arquivo, por exemplo, o sistema operacional é invocado para executar a chamada. O software independente de dispositivo olha no *cache* de blocos, por exemplo. Se o bloco necessário não estiver aí, ele chama o *driver* de dispositivo para enviar a solicitação ao hardware. O processo, então, é bloqueado até que a operação de disco seja concluída.

Quando o disco termina, o hardware, gera uma interrupção. O manipulador de interrupções é executado para descobrir o que aconteceu, isto é, qual dispositivo quer atenção imediatamente. Então, ele extrai o status do dispositivo e acorda o processo adormecido para terminar a solicitação de E/S e deixar o processo de usuário continuar.

3.3 IMPASSES

Os sistemas de computador estão repletos de recursos que podem ser utilizados apenas por um processo por vez. Exemplos comuns incluem plotadoras, leitores de CD-ROM, gravadores de CD-ROM, sistemas de backup em unidade de fita DAT 8mm e entradas na tabela de processos do sistema. Ter dois processos simultaneamente gravando na impressora resulta em uma confusão. Ter dois processos que utilizam a mesma entrada na tabela de processos provavelmente levará a uma queda do sistema. Portanto, todos os sistemas operacionais têm a capacidade de (temporariamente) conceder acesso exclusivo a certos recursos para um processo.

Para muitos aplicativos, um processo requer acesso exclusivo não a um, mas a vários recursos. Considere, por exemplo, uma empresa de marketing especializada em fazer grandes e detalhados mapas demográficos do país em uma plotadora de 1m de largura. As informações demográficas vêm de CD-ROMS contendo o censo e outros dados. Suponha que o processo *A* solicita a unidade de CD-ROM e obtém-na. Um momento mais tarde, o processo *B* solicita a plotadora e obtém-na também. Agora o processo *A* solicita a plotadora e bloqueia, esperando por ela. Por fim, o processo *B* solicita a unidade de CD-ROM e também bloqueia. Neste ponto, os dois processos estão bloqueados e permanecerão assim eternamente. Essa situação é chamada **impasse** (***deadlock***). Os impasses não são uma boa coisa para ter em seu sistema.

Os impasses podem ocorrer em muitas situações além dessa de solicitar dispositivos dedicados de E/S. Em um sistema de banco de dados, por exemplo, um programa pode precisar travar vários registros que ele está utilizando, para evitar condições de corrida. Se o processo *A* trava o registro *R1* e o processo *B* trava o registro *R2*, e cada processo, en-

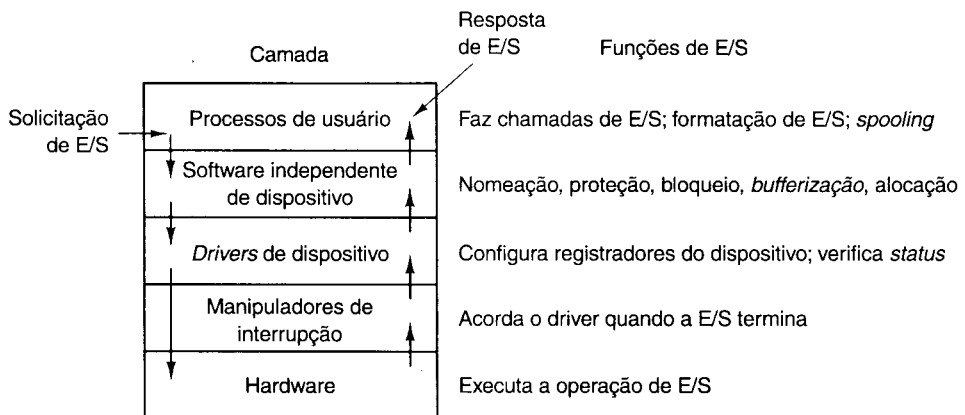



Figura 3-6 As camadas do sistema de E/S e as principais funções de cada uma.

tão, tenta travar o registro do outro, também temos um impasse. Os impasses, portanto, podem ocorrer em recursos de hardware ou em recursos de software.

Nesta seção, examinaremos impasses mais de perto para ver como eles surgem e como podem ser prevenidos ou evitados. Como exemplos, discutiremos a aquisição de dispositivos físicos como unidades de fita, unidades de CD-ROM e plotadoras, porque tais dispositivos são fáceis de visualizar, mas os princípios e os algoritmos aplicam-se igualmente bem a outros tipos de impasses.

3.3.1 Recursos

Os impasses podem ocorrer quando se concede aos processos acesso exclusivo a dispositivos, a arquivos, etc. Para tornar a discussão sobre impasses o mais geral possível, vamos referir-nos aos objetos concedidos como **recursos.** Um recurso pode ser um dispositivo de hardware (p. ex., uma unidade de fita) ou um conjunto de informações (p. ex., um registro bloqueado em um banco de dados). Um computador normalmente terá muitos recursos diferentes que podem ser adquiridos. Para alguns recursos, vários exemplares idênticos podem estar disponíveis, como três unidades de fita. Quando várias cópias de um recurso estão disponíveis, qualquer uma delas pode ser utilizada para satisfazer qualquer solicitação ao recurso. Em resumo, um recurso é qualquer coisa que pode ser utilizada somente por um único processo em qualquer instante.

Os recursos dividem-se em dois tipos: preemptível e não-preemptível. Um **recurso preemptível** é aquele que pode ser tirado do processo que é proprietário dele sem nenhum problema. A memória é um exemplo de um recurso preemptível. Considere, por exemplo, um sistema com 512K de memória de usuário, uma impressora e dois processos de 512K, cada um querendo imprimir algo. O processo *A* solicita e obtém a impressora, então, começa a calcular os valores a imprimir. Antes de finalizar os cálculos, ele excede seu quantum de tempo e é comutado para o disco.

O processo *B* agora executa e tenta, sem sucesso, obter a impressora. Potencialmente, agora temos uma situação de impasse, porque *A* tem a impressora e *B* tem a memória e nem um nem outro pode prosseguir sem o recurso segurado pelo outro. Felizmente, é possível preemptar (tirar) a memória de *B* comutando-a para o disco e comutando *A* para a memória. Agora *A* pode executar, fazer sua impressão e, então, liberar a impressora. Nenhum impasse ocorre.

Um **recurso não-preemptível,** em oposição, é aquele que não pode ser tirado de seu proprietário atual sem causar falha na computação. Se um processo começou a imprimir uma saída, a ação de tomar a impressora dele e dá-la a outro processo resultará em problemas na saída. As impressoras não são preemptíveis.

Em geral, impasses envolvem recursos não-preemptíveis. Impasses potenciais que envolvem recursos preemptíveis normalmente podem ser resolvidos mediante a realocação de recursos de um processo para outro. Assim, nosso tratamento irá concentrar-se em recursos não-preemptíveis.

A seqüência de eventos requerida para utilizar um recurso é:

1. Solicitar o recurso.
2. Utilizar o recurso.
3. Liberar o recurso.

Se o recurso não estiver disponível quando for solicitado, o processo solicitante é forçado a esperar. Em alguns sistemas operacionais, o processo é automaticamente bloqueado quando uma solicitação de recurso falha e é acordado quando o recurso torna-se disponível. Em outros sistemas, a solicitação falha com um código de erro e cabe ao processo de chamada esperar alguns instantes e tentar novamente.

3.3.2 Princípios Básicos de Impasses

O impasse pode ser definido formalmente como segue:

Um conjunto de processos está em um impasse se cada processo no conjunto está esperando um evento que somente outro processo no conjunto pode causar.

Como todos os processos estão esperando, nenhum deles jamais causará qualquer dos eventos que poderiam acordar qualquer dos outros membros do conjunto e todos os processos continuam a esperar eternamente.

Na maioria dos casos, o evento que cada processo está esperando é a liberação de algum recurso atualmente possuído por outro membro do conjunto. Em outras palavras, cada membro do conjunto de processos em impasse está esperando um recurso que é possuído por um processo em estado de impasse. Nenhum dos processos pode executar, nenhum deles pode liberar qualquer recurso e nenhum deles pode ser acordado. O número de processos e o número e o tipo de recursos possuídos e requeridos não têm importância.

Condições para um Impasse

Coffman e colaboradores (1971) demonstraram que quatro condições devem ser sustentadas para haver um impasse:

1. Condição de exclusão mútua. Todo recurso está atribuído a exatamente um processo ou está disponível.
2. Condição de segura e espera. Os processos que estão segurando recursos concedidos anteriormente podem solicitar novos recursos.
3. Condição de nenhuma preempção. Os recursos previamente concedidos não podem ser tirados de um processo. Eles devem ser explicitamente liberados pelo processo que os está segurando.
4. Condição de espera circular. Deve haver uma cadeia circular de dois ou mais processos, cada um dos quais está esperando um recurso segurado pelo próximo membro da cadeia.

Todas essas quatro condições devem estar presentes para um impasse ocorrer. Se uma ou mais dessas condições estiver ausente, nenhum impasse é possível.

Modelamento de Impasse

Holt (1972) demonstrou como essas quatro condições podem ser modeladas utilizando grafos dirigidos. Os grafos têm dois tipos de nós: processos, mostrados como círculos, e recursos, mostrados como quadrados. Um arco de um nó de recurso (quadrado) para um nó de processo (círculo) significa que o recurso previamente foi solicitado por, concedido para, e atualmente está sendo segurado por esse processo. Na Figura 3-7(a), o recurso *R* está atualmente atribuído ao processo *A*.

Um arco de um processo para um recurso significa que o processo atualmente está bloqueado esperando esse recurso. Na Figura 3-7(b), o processo *B* está esperando o recurso *S*. Na Figura 3-7(c) vemos um impasse: o processo *C* está esperando o recurso *T*, que atualmente está sendo segurado pelo processo *D*. O processo *D* não irá liberar o recurso *T* porque está esperando o recurso *U*, que está sendo segurado por *C*. Os dois processos esperarão eternamente. Um ciclo no grafo significa que há um impasse envolvendo os processos e os recursos no ciclo. Nesse exemplo, o ciclo é *C-T-D-U-C*.

Agora vamos olhar um exemplo de como grafos de recurso podem ser utilizados. Imagine que temos três processos, *A*, *B* e *C* e três recursos, *R*, *S* e T. As solicitações e as liberações dos três processos são dadas na Figura 3-8(a)-(c) O sistema operacional é livre para executar qualquer processo desbloqueado a qualquer instante, portanto, ele poderia decidir executar *A* até que *A* termine todo seu trabalho, depois executar *B* até a conclusão e por fim executar *C*.

Essa ordem não conduz a nenhum impasse (porque não há nenhuma competição por recursos), mas também não tem qualquer paralelismo. Além de solicitar e de liberar recursos, os processos computam e fazem E/S. Quando os processos são executados seqüencialmente não há nenhuma possibilidade de que, enquanto um processo está esperando E/S, outro possa utilizar a CPU. No entanto, executar os processos precisamente na seqüência pode não ser ótimo. Por outro lado, se nenhum dos processos fizer alguma E/S, o *job* mais curto primeiro é melhor que *round robin*, então, sob algumas circunstâncias executar todos os processos seqüencialmente pode ser o melhor caminho.

Agora vamos supor que os processos fazem tanto E/S como computações, de modo que o *round robin* torne-se um algoritmo de agendamento aceitável. As solicitações de recurso podem ocorrer na ordem da Figura 3-8(d). Se essas seis solicitações forem executadas nessa ordem, os seis grafos de recurso resultantes são mostrados na Figura 3-8(e)-(j). Depois que a solicitação 4 foi feita, *A* bloqueia esperando *S*, como mostrado na Figura 3-8(h). Nos próximos dois passos *B* e *C* também bloqueiam, conduzindo a um ciclo e ao impasse da Figura 3-8(j)

Entretanto, como já mencionamos, não se exige que o sistema operacional execute os processos em qualquer ordem especial. Em particular, se a concessão de uma solicitação particular pode levar a um impasse, o sistema operacional pode simplesmente suspender o processo sem conceder a solicitação (i. e., simplesmente não agenda o processo) até que ele esteja seguro. Na Figura 3-8, se o sistema operacional soubesse sobre o impasse prestes a acontecer, ele poderia suspender *B* em vez de conceder-lhe *S*. Executando somente *A* e *C*, obteríamos as solicitações e as liberações da Figura 3-8(k) em vez da Figura 3-8(d). Essa seqüência leva aos grafos de recurso da Figura 3-8(l)-(q), que não leva a impasse.

Após o passo (q), o processo *B* pode receber *S* porque *A* terminou e *C* tem tudo que precisa. Mesmo que *B* acabasse bloqueando ao solicitar *T*, nenhum impasse poderia ocorrer. *B* somente irá espera até que *C* termine.

Mais adiante neste capítulo, estudaremos um algoritmo detalhado para fazer decisões de alocação que não levam a um impasse. O ponto a entender agora é que grafos de recurso são uma ferramenta que permite ver se uma dada seqüência de solicitação/liberação leva a impasse. Nós simplesmente executamos as solicitações e as liberações passo a passo e depois de cada passo verificamos o grafo para ver se ele contém qualquer ciclo. Se contiver, temos um impasse; caso contrário, não há impasse. Embora nosso tratamento de grafos de recurso foi para o caso de um

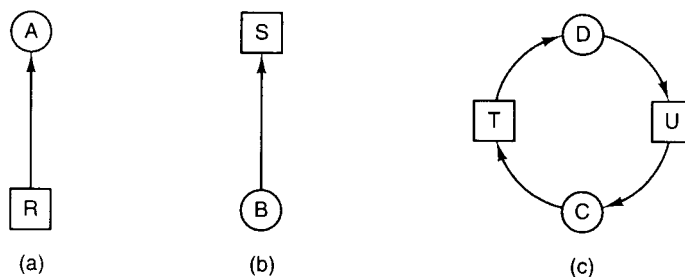



Figura 3-7 Grafos de alocação de recursos. (a) Segurando um recurso. (b) Solicitando um recurso. (c) Impasse.

único recurso de cada tipo, grafos de recursos também podem ser generalizados para tratar múltiplos recursos do mesmo tipo (Holt, 1972).

Em geral, quatro estratégias são utilizadas para lidar com impasses.

1. Simplesmente ignorar o problema.
2. Detecção e recuperação.
3. Impedimento dinâmico por cuidadosa alocação de recursos.
4. Prevenção, pela negação estrutural de uma das quatro condições necessárias.

Examinaremos cada um desses métodos nas próximas quatro seções.

3.3.3 O Algoritmo do Avestruz

A abordagem mais simples é o algoritmo do avestruz: enfie sua cabeça na areia e finja que não há nenhum problema. Pessoas diferentes reagem a essa estratégia de maneiras diferentes. Os matemáticos consideram-na totalmente inaceitável e dizem que impasses devem ser evitados a todo custo. Os engenheiros perguntam sobre a freqüência esperada do problema, sobre a freqüência das quedas de sistema por outras razões e sobre a gravidade do impasse. Se os impasses ocorressem à média de uma vez a cada 50 anos, mas quedas do sistema devido a falhas de hardware, erros de compilador e *bugs* do sistema operacional ocorrem uma vez por mês, a maioria dos engenheiros não estaria disposta a pagar um preço tão alto no desempenho ou na conveniência para eliminar impasses.

Para tornar esse contraste mais específico, o UNIX (e o MINIX) potencialmente passa por impasses que não são nem mesmo detectados, para não dizer automaticamente interrompidos. O número total de processos no sistema é determinado pelo número de entradas na tabela de processos. Assim as entradas da tabela de processos são recursos finitos. Se um FORK falha porque a tabela está cheia, uma abordagem razoável para o programa que faz o FORK é esperar um tempo aleatório e tentar novamente.

Agora suponha que um sistema UNIX tem 100 entradas de processos. Dez programas estão executando, cada um dos quais necessita criar 12 (sub)processos. Depois que cada processo criou 9 processos, os 10 processos originais e os 90 novos processos esgotaram a tabela. Cada um dos 10 processos originais agora fica em um laço interminável de criação e de falha — um impasse. A probabilidade desse evento é ínfima, mas ele *pode* acontecer. Deveríamos abandonar processos e a chamada FORK para eliminar o problema?

O número máximo de arquivos abertos é de maneira semelhante restringido pelo tamanho da tabela de nós-i; então, um problema semelhante ocorre quando ela é completamente preenchida. Espaço de troca (*swap*) em disco é outro recurso limitado. De fato, quase todas as tabelas no sistema operacional representam um recurso finito. Deveríamos abolir tudo isso porque talvez aconteça de uma coleção de n processos reivindicar $1/n$ do total para cada um e, então, todos tentarem reivindicar algo mais?

A abordagem do UNIX é simplesmente ignorar o problema na suposição de que a maioria dos usuários preferiria um impasse ocasional a uma regra que restringisse todos os usuários a um processo, a um arquivo aberto e "tudo" um. Se os impasses pudessem ser eliminados livremente, não haveria muita discussão. O problema é que o preço é alto, principalmente em termos de impor restrições inconvenientes em processos, como veremos brevemente. Assim nos defrontamos com uma negociação desagradável entre conveniência e correção e muita discussão sobre o que é mais importante.

3.3.4 Detecção e Recuperação

Uma segunda técnica é a detecção e recuperação. Quando essa técnica é utilizada, o sistema não faz nada exceto monitorar as solicitações e as liberações. Cada vez que um recurso é solicitado ou liberado, o grafo de recurso é atualizado, e uma verificação é feita para ver se qualquer ciclo existe. Se um ciclo existir, um dos processos no ciclo é eliminado. Se isso não quebrar o impasse, outro processo é eliminado e assim por diante até que o ciclo seja quebrado.

Um método relativamente mais rudimentar é nem mesmo manter o grafo de recurso, mas, em vez disso, verificar periodicamente se há qualquer processo que foi continuamente bloqueado por mais que, digamos, 1 hora. Esses processos, então, são eliminados.

A detecção e recuperação é a estratégia freqüentemente utilizada em grandes computadores *mainframe*, especialmente sistemas em lote nos quais eliminar um processo e, então, reiniciar é normalmente aceitável. Mas deve-se tomar cuidado de restaurar qualquer arquivo modificado ao seu estado original e desfazer qualquer outro efeito colateral que possa ter ocorrido.

3.3.5 Prevenção de Impasses

A terceira estratégia de impasse é impor restrições convenientes para os processos de tal modo que os impasses sejam estruturalmente impossíveis. As quatro condições declaradas por Coffman e colaboradores (1971) oferecem um indício para algumas possíveis soluções. Se pudermos assegurar que pelo menos uma dessas condições nunca seja satisfeita, então os impasses serão impossíveis (Havender, 1968).

Primeiro abordaremos a condição de exclusão mútua. Se nenhum recurso fosse atribuído exclusivamente a um único processo, nunca teríamos impasses. Entretanto, é igualmente claro que permitir dois processos gravar na impressora ao mesmo tempo levará ao caos. Fazendo *spool* da saída da impressora, vários processos podem gerar saída ao mesmo tempo. Nesse modelo, o único processo que

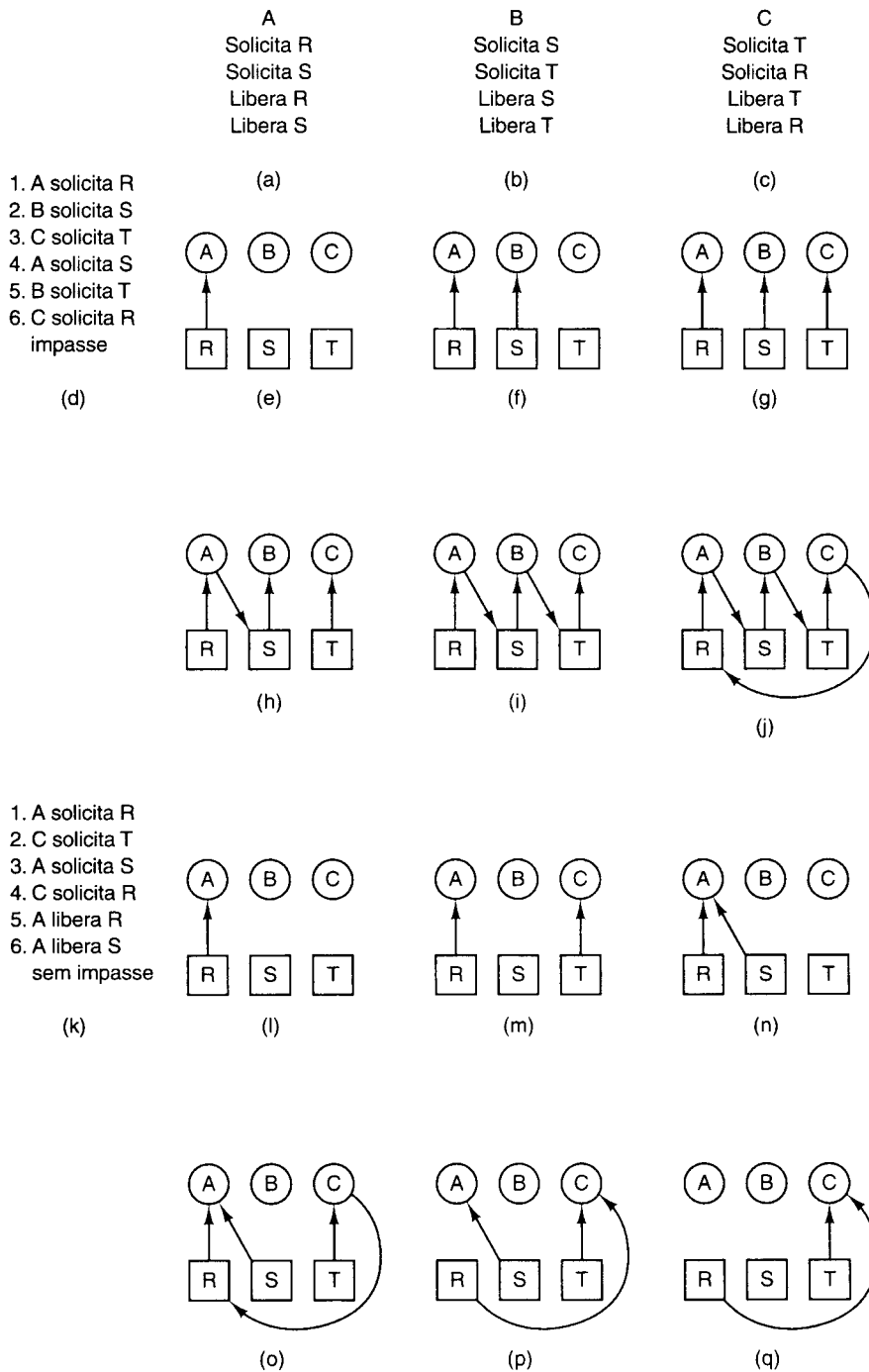



Figura 3-8 Um exemplo de como ocorre um impasse e de como ele pode ser evitado.

realmente solicita a impressora física é o *daemon* de impressora. Como o *daemon* nunca solicita qualquer outro recurso, podemos eliminar o impasse para a impressora

Infelizmente, nem todos os dispositivos podem ser enviados para o *spool* (a tabela de processos é um exemplo). Além disso, a competição por espaço em disco para fazer *spool* pode levar a um impasse. O que aconteceria se dois processos preenchessem, cada um, metade do espaço disponível para *spool* com saída e nenhum terminasse? Se o *daemon* fosse programado para começar a impressão até mesmo antes de toda a saída ser colocada no *spool*, a impressora poderia ficar desocupada se um processo de saída decidisse esperar várias horas após a primeira rajada de saída. Por essa razão, os *daemons* normalmente são programados para imprimir somente depois que o arquivo completo de saída está disponível. Nenhum processo jamais acabará; então, temos um impasse no disco.

A segunda das condições declarada por Coffman e colaboradores parece mais promissora. Se nos for possível prevenir que os processos que seguram recursos esperem mais recursos, poderíamos eliminar impasses. Uma maneira de alcançar essa meta é requerer que todos os processos solicitem todos os seus recursos antes de iniciar a execução. Se tudo estivesse disponível, o processo seria alocado sempre que necessário e poderia executar até sua conclusão. Se um ou mais recursos estivessem ocupados, nada seria alocado, e o processo simplesmente esperaria.

Um problema imediato com essa abordagem é que muitos processos não sabem de quantos recursos eles necessitarão até que tenham começado a executar. Outro problema é que os recursos não serão utilizados otimamente com essa abordagem. Tome, como um exemplo, um processo que lê dados de uma fita de entrada, analisa-os durante um hora e, então, grava uma fita de saída assim como plota os resultados. Se todos os recursos devessem ser requeridos antecipadamente, o processo amarraria a unidade de fita de saída e a plotadora durante uma hora.

Uma maneira ligeiramente diferente de quebrar a condição de segura e espera é requerer que um processo que solicita um recurso primeiro libere temporariamente todos os recursos atualmente sendo segurados. Somente se a solicitação for bem-sucedida ela poderá receber de volta os recursos originais.

Abordar a terceira condição (nenhuma preempção) é ainda menos promissor que abordar a segunda. Se a um processo obteve acesso a impressora e ele está no meio da impressão de sua saída, tomar à força a impressora, porque uma plotadora necessária não está disponível, resultará em uma confusão.

Restou somente uma condição. A espera circular pode ser eliminada de várias maneiras. Uma maneira é simplesmente ter uma regra que diz que um processo é intitulado somente para um único recurso em qualquer momento. Se precisar de um segundo, ele deve liberar o primeiro. Para um processo que necessita copiar um arquivo enorme de uma fita para uma impressora, essa restrição é inaceitável.

Outra maneira de evitar a espera circular é oferecer uma numeração global de todos os recursos, como mostrado na Figura 3-9(a). Agora a regra é esta: processos podem solicitar recursos sempre que quiserem, mas todas as solicitações devem ser feitas em ordem numérica. Um processo pode solicitar primeiro uma impressora e, então, uma unidade de fita, mas não pode solicitar primeiro uma plotadora e, então, uma impressora.

Com essa regra, o grafo de alocação de recursos nunca pode ter círculos. Deixe-nos ver por que isso é verdadeiro para o caso de dois processos, na Figura 3-9(b). Podemos obter um impasse somente se A solicita o recurso j e B solicita o recurso i. Supondo que i e j sejam recursos distintos, eles terão números diferentes. Se $i > j$, então A não tem permissão para solicitar j. Se $i < j$, então B não tem permissão para solicitar i. De qualquer maneira, o impasse é impossível.

Com múltiplos processos a mesma lógica mantém-se. Em cada instante, um dos recursos atribuídos será o mais alto. O processo que segura esse recurso nunca pedirá um recurso já atribuído. Ele terminará, ou na pior das hipóteses, solicitará recursos numerados ainda mais alto, todos os quais estão disponíveis. Por fim, ele terminará e liberará seus recursos. Nesse ponto, algum outro processo segurará o recurso mais alto e também pode terminar. Em resumo, aí existe um cenário em que todos os processos terminam, então, nenhum impasse está presente.

Uma variação menor desse algoritmo é derrubar o requisito de que os recursos devem ser adquiridos em uma seqüência precisamente crescente e simplesmente insistir que nenhum processo solicite um recurso mais baixo do que aquele que ele já está segurando. Se um processo inicialmente solicitar 9 e 10 e, então, liberar ambos, ele está efetivamente iniciando tudo de novo, então, não há nenhuma razão para proibi-lo de agora solicitar o recurso 1.

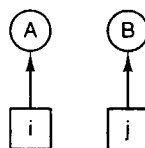



Figura 3-9 (a) Recursos numericamente ordenados. (b) Um grafo de recurso.

Condição	Abordagem
Exclusão mútua	Fazer spool de tudo
Segura e espera	Solicitar todos os recursos inicialmente
Nenhuma preempção	Tirar os recursos
Espera circular	Ordenar os recursos numericamente

Figura 3-10 Resumo das abordagens para prevenção de impasses.

Embora ordenar numericamente os recursos elimine o problema dos impasses, pode ser impossível descobrir uma ordem que satisfaça a todos. Quando os recursos incluem entradas na tabela de processos, espaço de *spool* em disco, registros bloqueados de banco de dados e outros recursos abstratos, o número de potenciais recursos e as diferentes utilizações podem ser tão grandes que talvez nenhuma ordenação funcione.

As várias abordagens para prevenção de impasses são resumidas na Figura 3-10.

3.3.6 Impedimento de Impasses

Na Figura 3-8 vimos que o impasse não foi evitado pela imposição de regras arbitrárias aos processos, mas pela análise cuidadosa de cada solicitação de recurso para ver se ele poderia ser concedido seguramente. A pergunta surge: há um algoritmo que sempre pode evitar impasses fazendo a escolha certa todas as vezes? A resposta é um qualificado sim — podemos evitar impasses, mas somente se certas informações estiverem disponíveis de antemão. Nesta seção, examinamos maneiras de evitar impasses mediante a alocação cuidadosa de recursos.

Algoritmo do Banqueiro para um Único Recurso

Um algoritmo de agendamento que pode evitar impasses é creditado a Dijkstra (1965) e conhecido como **algoritmo do banqueiro**. Ele é modelado na maneira como um banqueiro de um pequeno povoado poderia lidar com um grupo de clientes para os quais ele concedeu linhas de crédito. Na Figura 3-11(a) vemos quatro clientes, a cada um dos quais foi concedido um certo número de unidades de crédito (p. ex., 1 unidade é 1K dólares). O banqueiro sabe que nem todos os clientes precisarão do seu crédito máximo imediatamente; então, ele somente reservou 10 unidades em vez de 22 para atendê-los. (Nesta analogia, os clientes são os processos, as unidades são, digamos, as unidades de fita, e o banqueiro é o sistema operacional.)

Os clientes iniciam seus respectivos negócios, solicitando empréstimos de vez em quando. Em um certo momento, a situação é como a mostrada na Figura 3-11(b). Uma lista de clientes mostrando o dinheiro já emprestado (unidades de fita já atribuídas) e o crédito máximo disponível (número máximo de unidades de fita necessárias de uma vez mais tarde) é chamada **estado** do sistema com relação à alocação de recurso.

Um estado é conhecido como **seguro** se existir uma seqüência de outros estados que leva a todos os clientes a solicitarem empréstimos até seus limites de crédito (todos os processos obtêm todos os seus recursos e terminam.) O estado da Figura 3-11(b) é seguro porque, com duas unidades, o banqueiro pode adiar qualquer solicitação, exceto a de Marvin, deixando, assim, Marvin terminar e liberar todos os seus quatro recursos. Com quatro unidades em mãos, o banqueiro pode permitir que Suzanne ou Bárbara tenham as unidades necessárias, etc.

(a)

Nome	Usado	Máximo
Andy	0	6
Barbara	0	5
Marvin	0	4
Suzanne	0	7

Disponível: 10

(b)

Nome	Usado	Máximo
Andy	1	6
Barbara	1	5
Marvin	2	4
Suzanne	4	7

Disponível: 2

(c)

Nome	Usado	Máximo
Andy	1	6
Barbara	2	5
Marvin	2	4
Suzanne	4	7

Disponível: 1

Figura 3-11 Três estados de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

Considere o que aconteceria se uma solicitação de Barbara para mais uma unidade fosse concedida na Figura 3-11(b). Teríamos a situação da Figura 3-11(c), que é insegura. Se todos os clientes repentinamente solicitassem seus empréstimos máximos, o banqueiro não poderia satisfazer nenhum deles e teríamos um impasse. Um estado inseguro não conduz *necessariamente* a um impasse, desde que um cliente pode não precisar de toda a linha de crédito disponível, mas o banqueiro não pode confiar nesse comportamento.

Portanto, o algoritmo do banqueiro serve para considerar cada solicitação conforme ela ocorre e ver se o fato de atendê-la conduz a um estado seguro. Se conduzir, a solicitação é atendida: caso contrário, é postergada. Para ver se um estado é seguro, o banqueiro verifica se tem recursos suficientes para satisfazer o cliente o mais perto do máximo exigido por esse cliente. Se conseguir, ele assume que esses empréstimos serão pagos, e o cliente agora mais perto do seu limite é verificado e assim por diante. Se todos os empréstimos eventualmente puderem ser pagos, o estado é seguro e a solicitação inicial pode ser concedida.

Trajetórias de Recursos

O algoritmo acima foi descrito em termos de uma única classe de recursos (p. ex., somente unidades de fita ou somente impressoras, mas não um pouco de cada). Na Figura 3-12, vemos um modelo para lidar com dois processos e com dois recursos, por exemplo, uma impressora e uma plotadora. O eixo horizontal representa o número de instruções executadas pelo processo *A*. O eixo vertical representa o número de instruções executadas pelo processo *B*. Em I_1, *A* solicita uma impressora; em I_2, ele precisa de uma plotadora. A impressora e a plotadora são liberadas em I_3 e I_4, respectivamente. O processo *B* precisa da plotadora de I_5 até I_7 e da impressora de I_6 até I_8.

Cada ponto no diagrama representa um estado de união de dois processos. Inicialmente, o estado está em *p*, com nenhum processo tendo executado qualquer instrução. Se o agendador escolher executar *A* primeiro, alcançamos o ponto *q* em que *A* executou um certo número de instruções, mas *B* não executou nenhuma. No ponto *q*, a trajetória torna-se vertical que o agendador escolheu executar *B*. Com um único processador, todos os caminhos devem ser horizontais ou verticais, nunca diagonais. Além disso, o movimento é sempre para o norte ou para o leste, nunca para o sul ou para o oeste (os processos não podem executar para trás).

Quando *A* cruza a linha I_1 no caminho de *r* para *s*, ele solicita e lhe é concedida a impressora. Quando *B* alcança o ponto *t*, ele solicita a plotadora.

As regiões sombreadas são sobremaneira interessantes. As regiões com linhas inclinadas de sudoeste para nordeste representam que dois processos têm a impressora. A regra da exclusão mútua torna impossível entrar nessa região. De maneira semelhante, a região sombreada no sentido contrário representa que os dois processos têm a plotadora e é igualmente impossível.

Se o sistema jamais entrar na caixa delimitada por I_1 e I_2 nos lados e I_5 e I_6 nas partes superior e inferior, ele acabará em um impasse quando chegar à interseção de I_2 e I_6. Nesse ponto, *A* está solicitando a plotadora, e *B* está solicitando a impressora, mas ambas já foram atribuídas. A caixa inteira é insegura e não se deve entrar nela. No ponto *t*, a única coisa segura a fazer é executar o processo *A* até chegar a I_4. Além desse ponto, qualquer trajetória até *u* servirá.

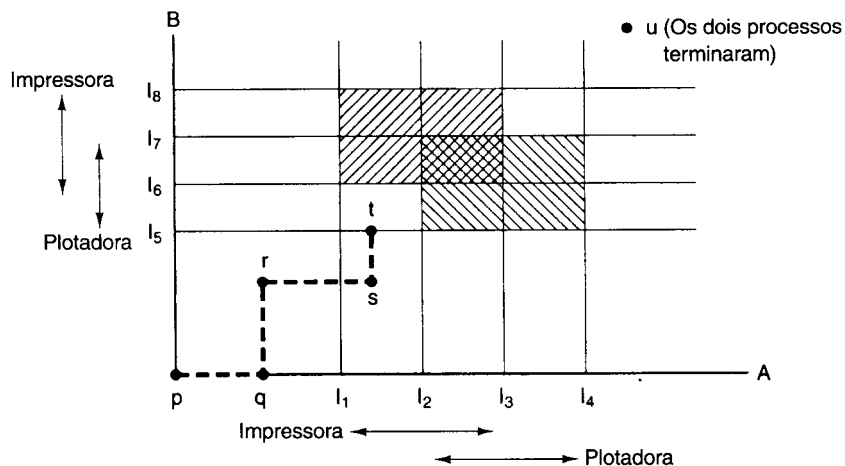



Figura 3-12 Duas trajetórias de recurso de processo.

O Algoritmo do Banqueiro para Múltiplos Recursos

Esse modelo gráfico é difícil de aplicar para o caso geral de um número arbitrário de processos e para um número arbitrário de classes de recursos, cada um com múltiplas instâncias (p. ex., duas plotadoras, três unidades de fita). Entretanto, o algoritmo do banqueiro pode ser generalizado para fazer o trabalho. A Figura 3-13 mostra como ele funciona.

Na Figura 3-13 vemos duas matrizes. A da esquerda mostra quanto de cada recurso está atualmente atribuído a cada um dos cinco processos. A matriz à direita mostra quantos recursos cada processo ainda precisa de modo que possa completar-se. Como no caso de um recurso único, os processos devem declarar suas necessidades totais de recurso antes de executar, para que o sistema possa calcular a matriz da direita em cada passo.

Os três vetores à direita da figura mostram os recursos existentes, *E*, os recursos possuídos, *P*, e os recursos disponíveis, *A*, respectivamente. A partir de *E* vemos que o sistema tem seis unidades de fita, três plotadoras, quatro impressoras e dois CD-ROMs. Desses, cinco unidades de fita, três plotadoras, duas impressoras e dois CD-ROMs atualmente estão atribuídos. Esse fato pode ser visto adicionando-se as quatro colunas de recursos na matriz esquerda. O vetor de recursos disponíveis é simplesmente a diferença entre o que o sistema tem e o que está atualmente em utilização.

O algoritmo para verificar se um estado é seguro agora pode ser declarado.

1. Procure uma linha, *R*, cujas necessidades de recursos não-atendidas são todas menores que ou iguais a *A*. Se não existir essa linha, o sistema acabará em um impasse uma vez que nenhum processo pode executar até sua conclusão.
2. Suponha que o processo da linha escolhida solicite todos os recursos que precisa (o que é garantido que é possível) e termine. Marque esse processo como terminado e adicione todos os seus recursos ao vetor *A*.
3. Repita os passos 1 e 2 até que todos os processos estejam marcados como terminado, caso em que o estado inicial era seguro, ou até que um impasse ocorra, caso em que não era seguro.

Se vários processos são elegíveis para serem escolhidos no passo 1, não importa qual é selecionado: o *pool* de recursos aumenta, ou no pior caso, permanece o mesmo.

Agora voltemos ao exemplo da Figura 3-13. O estado atual é seguro. Suponha que o processo *B* agora solicite uma impressora. Essa solicitação pode ser concedida porque o estado resultante ainda é seguro (o processo *D* pode terminar, e, então, dar vez aos processos *A* ou *E*, seguidos pelo resto).

Agora imagine que depois de dar a *B* uma das duas impressoras que restaram, *E* queira ter a última impressora. Conceder essa solicitação reduziria o vetor dos recursos disponíveis para (1 0 0 0), o que conduz a um impasse. É claro que a solicitação de *E* não pode ser satisfeita imediatamente e deve ser adiada temporariamente.

Esse algoritmo foi publicado pela primeira vez por Dijkstra em 1965. Desde então, quase todos os livros sobre sistemas operacionais descreveram-no em detalhe. Inumeráveis trabalhos foram escritos sobre vários aspectos dele. Infelizmente, poucos autores tiveram a coragem de indicar que embora na teoria o algoritmo seja maravilhoso, na prática ele é essencialmente inútil porque os processos raramente sabem quais são suas necessidades máximas de recursos previamente. Além do mais, o número de processos não é fixo, mas dinamicamente variável conforme novos usuários se conectam e se desconectam. Além disso, os recursos que foram considerados como disponíveis repentinamente podem desaparecer (unidades de fita podem quebrar).

Em resumo, os esquemas descritos anteriormente sob o nome "prevenção" são muito restritivos, e o algoritmo des-

Recursos atribuídos

Processos	Unidades de fita	Plotadoras	Impressoras	CD-ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Recursos ainda necessários

Processos	Unidades de fita	Plotadoras	Impressoras	CD-ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Figura 3-13 O algoritmo do banqueiro com múltiplos recursos.

crito aqui como "impedimento" requer informações que normalmente não estão disponíveis. Se você pode imaginar um algoritmo de propósito geral que faça o trabalho na prática tão bem quanto na teoria, escreva-o e envie-o a uma publicação de ciência da computação.

Para aplicações específicas, são conhecidos muitos algoritmos excelentes de propósito especial. Como um exemplo, em muitos sistemas de banco de dados, uma operação que ocorre freqüentemente é requisitar bloqueios em vários registros e, então, atualizar todos os registros bloqueados. Quando múltiplos processos estão executando ao mesmo tempo, há um perigo real de impasse.

A abordagem freqüentemente utilizada é chamada bloqueio de **duas fases**. Na primeira fase, o processo tenta bloquear todos os registros que precisa, um por vez. Se tiver sucesso, ele executa suas atualizações e libera os bloqueios. Se algum registro já estiver bloqueado, ele libera os bloqueios que ele já tem e simplesmente começa tudo de novo. Em um certo sentido, essa abordagem é semelhante a solicitar todos os recursos necessários previamente ou pelo menos antes de qualquer coisa irrevogável ser feita.

Entretanto, essa estratégia não é aplicável de maneira geral. Em sistemas de tempo real e em sistemas de controle de processos, por exemplo, não é aceitável simplesmente terminar um processo no meio do caminho porque um recurso não está disponível e começar tudo de novo. Nem é aceitável começar tudo de novo se o processo leu ou gravou mensagens na rede, atualizou arquivos ou fez qualquer outra coisa que não pode ser repetida com segurança. O algoritmo funciona somente nas situações em que o programador muito cuidadosamente organizou as coisas de tal modo que o programa pode ser interrompido em qualquer ponto durante a primeira fase e reiniciado. Infelizmente, nem todos os aplicativos podem ser estruturados dessa maneira.

3.4 VISÃO GERAL DE E/S NO MINIX

A E/S do MINIX é estruturada como mostrado na Figura 3-6. As quatro primeiras camadas superiores dessa figura correspondem à estrutura de quatro camadas do MINIX mostrada na Figura 2-26. Nas seções a seguir, veremos um resumo de cada uma das camadas, com ênfase nos *drivers* de dispositivo. O tratamento de interrupções foi estudado no capítulo anterior, e a E/S independente do dispositivo será discutida quando abordarmos o sistema de arquivos, no Capítulo. 5.

3.4.1 Manipuladores de Interrupções no MINIX

Muitos dos *drivers* de dispositivo iniciam algum dispositivo de E/S e, então, bloqueiam, esperando uma mensagem chegar. Essa mensagem normalmente é gerada pelo manipulador de interrupções do dispositivo. Outros *drivers* de dispositivo não iniciam nenhuma E/S física (p. ex., ler de um disco de RAM e gravar em um dispositivo de exibição mapeado em memória), não utilizam interrupções e não esperam uma mensagem de um dispositivo de E/S. No capítulo anterior, o mecanismo por meio do qual as interrupções geram mensagens e causam comutação de tarefas foi apresentado em grande detalhe e não falaremos mais sobre ele aqui. Mas os manipuladores de interrupções podem fazer mais do que apenas gerar uma mensagem. Freqüentemente eles também fazem algum trabalho no processamento de entrada e de saída de baixo nível. Discutiremos isso de uma maneira geral aqui e, então, retornaremos aos detalhes quando abordarmos o código para vários dispositivos.

Para dispositivos de disco, a entrada e a saída é geralmente uma questão de comandar um dispositivo para executar sua operação e, então, esperar até que a operação esteja completa. A controladora de disco faz a maior parte do trabalho, e muito pouco é exigido do manipulador de interrupções. Vimos que o manipulador de interrupções inteiro para a tarefa de disco rígido consiste em somente três linhas de código, com a única operação de E/S sendo a leitura de um único byte para determinar o status da controladora. Nossa vida seria simples de fato se todas as interrupções pudessem ser tratadas assim tão facilmente.

Entretanto, às vezes, há mais coisas para o manipulador de baixo nível fazer. O mecanismo de passagem de mensagens tem um custo. Quando uma interrupção pode ocorrer freqüentemente mas a quantidade de E/S tratada pela interrupção é pequena, pode valer a pena fazer o próprio manipulador trabalhar um pouco mais e adiar o envio de uma mensagem para a tarefa até uma interrupção subseqüente, quando há mais para a tarefa fazer. O MINIX trata interrupções do relógio dessa maneira. Em muitos tiques de relógio há muito pouco a ser feito, exceto manter o tempo. Isso pode ser feito sem enviar uma mensagem à própria tarefa de relógio. O manipulador de relógio incrementa uma variável, apropriadamente chamada *pending_ticks*. O tempo atual é a soma do tempo registrado quando a própria tarefa de relógio executou pela última vez mais o valor de *pending_ticks*. Quando a tarefa de relógio recebe uma mensagem e acorda, ela adiciona *pending_ticks* à sua variável principal de monitoração do tempo e, então, zera *pending_ticks*. O manipulador de interrupções do relógio examina algumas outras variáveis e envia uma mensagem à tarefa de relógio somente quando detecta que a tarefa tem trabalho real a fazer, como entregar um alarme ou agendar um novo processo para executar. Ele também pode enviar uma mensagem à tarefa de terminal.

Na tarefa de terminal, vemos outra variação do tema manipulador de interrupções. Essa tarefa trata vários tipos diferentes de hardware, incluindo o teclado e as linhas RS-232. Cada um desses têm seu próprio manipulador de interrupções. O teclado ajusta-se exatamente na descrição de um dispositivo em que pode haver relativamente pouca E/S a fazer em resposta a cada interrupção. Em um PC, uma interrupção ocorre cada vez que uma tecla é pressionada ou é liberada. Isso inclui teclas especiais como as teclas SHIFT e CTRL. Mas, se as ignoramos por um momento,

poderemos dizer que, na média, metade de um caractere é recebido por interrupção. Como não há muito o que a tarefa de terminal possa fazer com metade um caractere, faz sentido enviar-lhe uma mensagem somente quando algo que vale a pena pode ser realizado. Examinaremos os detalhes mais tarde; por enquanto, diremos apenas que o manipulador de interrupções de teclado faz a leitura de baixo nível dos dados do teclado e, então, filtra eventos que pode ignorar, como a liberação de uma tecla comum. (A liberação de uma tecla especial, p. ex., a tecla SHIFT, não pode ser ignorada.) Então, os códigos que representam todos os eventos não-ignorados são colocados em uma fila para posterior processamento pela própria tarefa de terminal.

O manipulador de interrupções de teclado difere do paradigma simples que apresentamos do manipulador de interrupções que envia uma mensagem para sua tarefa associada, porque este manipulador de interrupções não envia absolutamente nenhuma mensagem. Em vez disso, quando adiciona um código à fila, ele modifica uma variável, *tty_timeout*, que é lida pelo manipulador de interrupções do relógio. Quando uma interrupção não muda a fila, *tty_timeout* também não é mudado. No próximo tique de relógio, o manipulador de relógio envia uma mensagem à tarefa de terminal se houver alterações na fila. Outros manipuladores de interrupções do tipo terminal, por exemplo, aqueles para as linhas RS-232, funcionam da mesma maneira. Uma mensagem para tarefa de terminal chegará logo depois que um caractere for recebido, mas uma mensagem não é necessariamente gerada para cada caractere quando os caracteres estão chegando rapidamente. Vários caracteres podem acumular e, então, podem ser processados em resposta a uma única mensagem. Além disso, todos os dispositivos terminais são verificados cada vez que uma mensagem é recebida pela tarefa de terminal.

3.4.2 *Drivers* de Dispositivo no MINIX

Para cada classe de dispositivo de E/S presente em um sistema MINIX, uma tarefa de E/S (*driver* de dispositivo) diferente está presente. Esses *drivers* são processos completos, cada um com seus próprios estados, registradores, pilhas e assim por diante. Os *drivers* de dispositivo comunicam-se entre si (onde necessário) e com o sistema de arquivos utilizando o mecanismo-padrão de passagem de mensagens utilizado por todos os processos do MINIX. *Drivers* de dispositivo simples estão escritos como arquivos-fonte únicos, como *clock.c*. Para outros *drivers*, como os para o disco de RAM, o disco rígido e o disquete, há um arquivo-fonte suportando cada tipo de dispositivo, assim como um conjunto de rotinas comuns em *driver.c* para suportar todos os diferentes tipos de hardware. Em um sentido, isso divide o nível de *driver* de dispositivo da Figura 3-6 em dois subníveis. Essa separação das partes do software dependente do hardware e independente do hardware facilita a adaptação para uma variedade de configurações de hardware. Embora algum código-fonte em comum seja utilizado, o *driver* para cada tipo de disco executa como um processo separado, para suportar rápidas transferências de dados.

De modo semelhante, o código-fonte do *driver* de terminal é organizado com o código independente do hardware em *tty.c* e o código-fonte para suportar dispositivos diferentes, tal como consoles mapeados de memória, teclado, linhas seriais e pseudoterminais em arquivos separados. Nesse caso, entretanto, um único processo suporta todos os diferentes tipos de dispositivo.

Para grupos de dispositivos como dispositivos de disco e de terminais, para os quais pode haver vários arquivos-fonte, também há arquivos de cabeçalho. *Driver.h* suporta todos os *drivers* de dispositivo de bloco. *Tty.h* oferece definições comuns para todos os dispositivos terminais.

A principal diferença entre *drivers* de dispositivo e outros processos é que os *drivers* de dispositivo são vinculados juntos no *kernel* e, assim, todos compartilham um espaço comum de endereço. Como resultado, se vários *drivers* de dispositivo utilizarem um procedimento comum, somente uma cópia será vinculada no código binário do MINIX.

Esse projeto é altamente modular e moderadamente eficiente. É também um dos poucos lugares onde o MINIX difere do UNIX de uma maneira essencial. No MINIX um processo lê um arquivo enviando uma mensagem para o processo de sistema de arquivos. O sistema de arquivos, por sua vez, pode enviar uma mensagem para o *driver* de disco solicitando que ele leia o bloco necessário. Essa seqüência (ligeiramente simplificada em relação ao que acontece na realidade) é mostrada na Figura 3-14(a). Fazendo essas interações via mecanismo de mensagens, forçamos várias partes do sistema a interfacear de maneiras padronizadas com outras partes. Contudo, colocando todos os *drivers* de dispositivo no espaço de endereço do *kernel*, eles têm acesso fácil à tabela de processos e a outras estruturas de dados-chave quando necessário.

No UNIX todos os processos têm duas partes: uma parte no espaço do usuário e uma parte no espaço do *kernel*, como mostrado na Figura 3-14(b) Quando uma chamada de sistema é feita, o sistema operacional alterna da parte no espaço do usuário para a parte no espaço do *kernel* de uma maneira algo mágica. Essa estrutura é um remanescente do projeto do MULTICS, na qual a comutação era apenas uma chamada de procedimento comum, em vez de uma interrupção seguida pelo salvamento do estado da parte do usuário, como é no UNIX.

Os *drivers* de dispositivo no UNIX são simplesmente procedimentos do *kernel* que são chamados pela parte no espaço de *kernel* do processo. Quando um *driver* precisa esperar uma interrupção, ele chama um procedimento do *kernel* que o coloca para dormir até que algum manipulador de interrupções acorde-o. Note que é o próprio processo de usuário que está sendo colocado para dormir aqui, porque as partes do *kernel* e do usuário são na realidade partes diferentes do mesmo processo.

Entre os projetistas de sistema operacional, argumentos sobre os méritos dos sistemas monolíticos, como no UNIX,

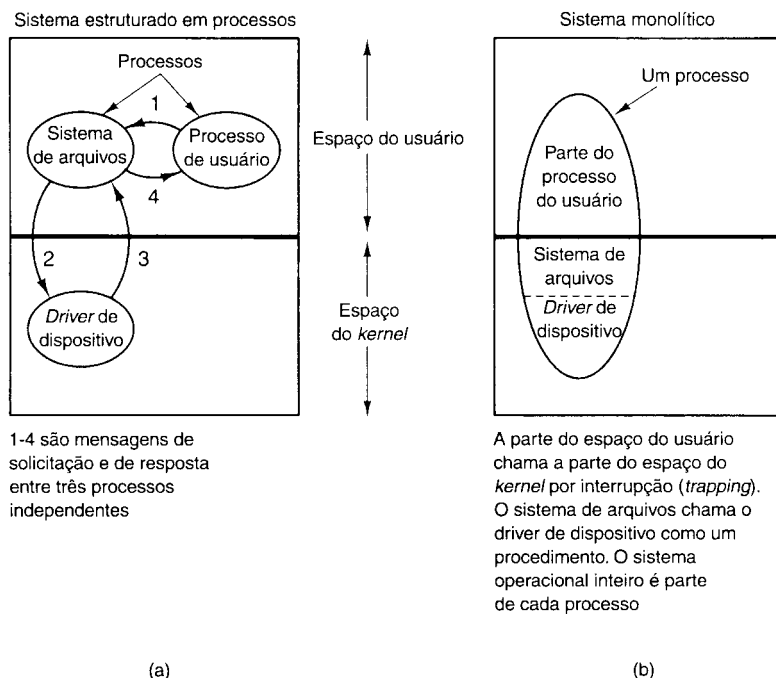



Figura 3-14 Duas maneiras de estruturar a comunicação sistema-usuário.

versus sistemas estruturados por processos, como no MINIX, são intermináveis. A abordagem do MINIX é melhor estruturada (mais modular), tem interfaces mais limpas entre as partes e estende-se facilmente para sistemas distribuídos em que os vários processos executam em computadores diferentes. A abordagem no UNIX é mais eficiente, porque as chamadas de procedimento são muito mais rápidas que o envio de mensagens. O MINIX foi dividido em muitos processos porque acreditamos que com computadores pessoais cada vez mais poderosos disponíveis, valeria a pena tornar o sistema ligeiramente mais lento para obter uma estrutura de software mais limpa. Mas leve em conta que muitos projetistas de sistema operacional não compartilham dessa crença.

Neste capítulo, discutimos *drivers* para disco de RAM, disco rígido, relógio e terminal. A configuração-padrão do MINIX também inclui *drivers* para disquete e para impressora, que não são discutidos em detalhe. A distribuição padrão do MINIX contém o código-fonte de *drivers* adicionais para linhas seriais RS-232, uma interface SCSI, CD-ROM, adaptador Ethernet e placa de som. Esses podem ser incluídos recompilando o MINIX.

Todas essas tarefas interfaceiam com outras partes do sistema MINIX da mesma maneira: mensagens de solicitação são enviadas para as tarefas. As mensagens contêm uma variedade de campos para armazenar o código de operação (p. ex., READ ou WRITE) e seus parâmetros. Uma tarefa tenta atender uma solicitação e retorna uma mensagem de resposta.

Para dispositivos de bloco, os campos das mensagens de solicitação e de resposta são mostrados na Figura 3-15. A mensagem de solicitação inclui o endereço de uma área de buffer que contém os dados a serem transmitidos ou na qual são esperados dados recebidos. A resposta inclui as informações de status para que o processo solicitante possa verificar se sua solicitação foi adequadamente executada. Os campos para os dispositivos de caractere são basicamente semelhantes mas podem variar ligeiramente de tarefa para tarefa. As mensagens para a tarefa de relógio, por exemplo, contêm tempos, e as mensagens para a tarefa de terminal podem conter o endereço de uma estrutura de dados que especifica todos os muitos aspectos configuráveis de um terminal, como os caracteres utilizados pelas funções de edição *erase-character* (apaga caractere) e *kill-line* (elimina linha).

A função de cada tarefa é aceitar solicitações de outros processos, normalmente o sistema de arquivos, e executá-las. Todas as tarefas de dispositivo de bloco foram escritas para receber uma mensagem, executá-la e enviar uma resposta. Entre outras coisas, essa decisão significa que essas tarefas são estritamente seqüenciais e não contém nenhuma multiprogramação interna, para mantê-las simples. Quando uma solicitação de hardware é feita, a tarefa faz uma operação RECEIVE especificando que está interessada

apenas em aceitar mensagens de interrupções, não novas solicitações para trabalhar. Quaisquer novas mensagens de solicitação são apenas mantidas esperando até que o trabalho atual tenha sido feito (princípio do *rendez-vous*). A tarefa de terminal é ligeiramente diferente, uma vez que uma única tarefa serve a vários dispositivos. Assim, é possível aceitar uma nova solicitação para entrada do teclado enquanto uma solicitação para ler uma linha serial ainda está sendo atendida. Contudo, para cada dispositivo uma solicitação deve ser completada antes de iniciar uma nova.

O programa principal para cada *driver* de dispositivo de bloco é estruturalmente o mesmo e é delineado na Figura 3-16. Quando o sistema inicia pela primeira vez, cada um dos *drivers* é inicializado, por sua vez, para dar a cada um a chance de inicializar tabelas internas e coisas semelhantes. Então, a tarefa de cada *driver* bloqueia, tentando obter uma mensagem. Quando uma mensagem chega, a identidade do processo que chama é salva, e o procedimento é chamado para executar o trabalho, com um procedimento diferente invocado para cada operação disponível. Depois que o trabalho terminou, uma resposta é enviada de volta para o processo que chamou, e a tarefa, então, volta para o topo do laço para esperar a próxima solicitação.

Cada um dos procedimentos *dev_xxx* trata uma das operações de que o *driver* é capaz. Ele retorna um código de status, informando o que aconteceu. O código de status, incluído na mensagem de resposta como o campo *REP_STATUS*, é a contagem de bytes transferidos (zero ou positiva) se tudo deu certo, ou o número do erro (negativo) se algo deu errado. Essa contagem pode diferir do número solicitado de bytes. Quando o fim de um arquivo é alcançado, o número de bytes disponível pode ser inferior ao número solicitado. Em terminais, no máximo uma linha é retornada, mesmo que a contagem solicitada seja maior.

3.4.3 Software de E/S Independente de Dispositivo no MINIX

No MINIX o processo do sistema de arquivos contém todo o código de E/S independente de dispositivo. O sistema de E/S está tão intimamente relacionado com o sistema de arquivos que eles foram fundidos em um processo. As funções realizadas pelo sistema de arquivos são mostradas na Figura 3-5, exceto pela solicitação e pela liberação de dispositivos dedicados, que não existem no MINIX como ele está atualmente configurado. Mas elas podem facilmente ser adicionadas aos *drivers* de dispositivo relevantes se a necessidade surgir no futuro.

Além do tratamento da interface com os *drivers*, *bufferização* e alocação de blocos, o sistema de arquivos também trata da proteção e de gerenciamento de diretórios, nós-i e sistemas de arquivos montados. Ele será abordado em detalhe no Capítulo 5.

Solicitações		
Campo	**Tipo**	**Significado**
m.m_type	int	Operação solicitada
m.DEVICE	int	Dispositivo secundário a utilizar
m.PROC_NR	int	Processo solicitando a E/S
m.COUNT	int	Contagem de *bytes* ou código ioctl
m. POSITION	long	Posição no dispositivo
m. ADRESS	char*	Endereço dentro do processo solicitante

Respostas		
Campo	**Tipo**	**Significado**
m.m_type	int	Sempre TASK_REPLY
m.PROC_REC_NR	int	O mesmo que PROC_NR na solicitação
m.REP_STATUS	Int	Bytes transferidos ou número do erro

Figura 3-15 Os campos das mensagens enviadas pelo sistema de arquivos para os *drivers* de dispositivo de bloco e os campos das respostas enviadas de volta.

```
message mess;                       /* buffer de mensagem */

void io_task() {
  initialize();                     /* feito só uma vez, durante a inicialização do sistema */
  while (TRUE) {
       receive(ANY, &mess);         /* espera uma solicitação para trabalhar */
       caller = mess.source;        /* processo de quem a mensagem veio */{
       switch(mess.type) {
           case READ:    rcode = dev_read(&mess); break;
           case WRITE:   rcode = dev_write(&mess), break;
           /* Outros casos entram aqui, incluindo OPEN, CLOSE e IOCTL */
           default:      rcode = ERROR;
       }
       mess.type = TASK_REPLY;
       mess.status = rcode;         /* código de resultado */
       send(caller, &mess);         /* envia de volta a mensagem de resposta para o processo */
  }
}
```

Figura 3-16 Esboço do procedimento principal de uma tarefa de E/S.

3.4.4 Software de E/S no Nível de Usuário no MINIX

O modelo geral delineado anteriormente neste capítulo também se aplica aqui. Procedimentos de biblioteca estão disponíveis para fazer chamadas de sistema e para todas as funções de C exigidas pelo padrão POSIX, como as funções de formatação de entrada e de saída *printf* e *scanf*. A configuração-padrão do MINIX contém um *daemon* para *spool*, *lpd*, que faz *spool* e imprime arquivos passados para ele pelo comando *lp*. A distribuição padrão do MINIX contém diversos *daemons* que suportam várias funções de rede. As operações de rede requerem algum suporte do sistema operacional que não é parte do MINIX na configuração descrita neste livro, mas o MINIX pode ser facilmente recompilado para adicionar o servidor de rede. Ele executa na mesma prioridade que o gerenciador de memória e o sistema de arquivos e, como eles, executa como um processo de usuário.

3.4.5 Manipulação de Impasses no MINIX

Fiel à sua herança, o MINIX segue o mesmo caminho que o UNIX com relação a impasses: ele apenas ignora o problema. O MINIX não contém dispositivos dedicados de E/S, embora se alguém quisesse pendurar uma unidade de fita DAT padrão da indústria em um PC, fazer o software para isso não representaria qualquer problema especial. Em resumo, o único lugar em que os impasses podem ocorrer são com os recursos implicitamente compartilhados, como as entradas da tabela de processos, as entradas da tabela de nós-i e assim por diante. Nenhum dos algoritmos de impasse conhecidos pode lidar com recursos como esses que não são solicitados explicitamente.

Realmente, o que foi dito acima não é estritamente verdadeiro. Aceitar o risco que processos de usuário poderiam cair em um impasse é uma coisa, mas dentro do próprio sistema operacional há alguns lugares em que se tomou um cuidado considerável para evitar problemas. O principal é a interação entre o sistema de arquivos e o gerenciador de memória. O gerenciador de memória envia mensagens para o sistema de arquivos ler o arquivo binário (programa executável) durante uma chamada de sistema EXEC, assim como em outros contextos. Se o sistema de arquivos não estiver desocupado quando o gerenciador de memória estiver tentando enviar para ele, o gerenciador de memória será bloqueado. Se o sistema de arquivos, então, precisasse tentar enviar uma mensagem para o gerenciador de memória, ele também descobriria que o *rendez-vous* falhou e bloquearia, levando a um impasse.

Esse problema foi evitado construindo o sistema de tal maneira que o sistema de arquivos nunca envia as mensagens de *solicitação* para o gerenciador de memória, somente *respostas*, com uma pequena exceção. A exceção é que ao iniciar, o sistema de arquivos informa o tamanho do disco de RAM para o gerenciador de memória, que seguramente está esperando a mensagem.

É possível bloquear dispositivos e arquivos mesmo sem suporte do sistema operacional. Um nome de arquivo pode servir como uma variável verdadeiramente global, cuja presença ou ausência pode ser notada por todos os outros processos. Um diretório especial, *usr/spool/locks/*, está normalmente presente nos sistemas MINIX, como na maioria dos sistemas UNIX, onde os processos podem criar **arquivos de bloqueio**, para marcar qualquer recurso que eles estejam utilizando. O sistema de arquivos MINIX também suporta o estilo de bloqueio de arquivo aconselhável do POSIX. Mas nenhum desses mecanismos é imposto. Eles dependem do bom comportamento dos processos e não há nada para impedir que um programa utilize um recurso que está bloqueado por outro processo. Isso não é exatamente a mesma coisa que preempção do recurso, porque não impede o primeiro processo de tentar continuar sua

utilização do recurso. Em outras palavras, não há exclusão mútua. O resultado de tal ação por um processo malcomportado é provavelmente uma confusão, mas não resulta em nenhum impasse.

3.5 DISPOSITIVOS DE BLOCO NO MINIX

Nas seções a seguir, retornaremos aos *drivers* de dispositivo, o tema principal deste capítulo e estudaremos vários deles detalhadamente. O MINIX suporta vários dispositivos de bloco diferentes. Então começaremos discutindo os aspectos comuns a todos os dispositivos de bloco. Discutiremos o disco de RAM, o disco rígido e o disquete. Cada um desses é interessante por uma razão diferente. O disco de RAM é um bom exemplo para estudar porque tem todas as propriedades dos dispositivos de bloco em geral, exceto a E/S real — porque o "disco" é, na realidade, somente uma parte da memória. Essa simplicidade torna-o um bom ponto de partida. O disco rígido mostra como deve ser um *driver* de disco real. Poderia esperar-se que o disquete fosse mais fácil de suportar que o disco rígido mas, na realidade, não é. Não discutiremos todos os detalhes do disquete, mas indicaremos várias das complicações encontradas no *driver* de disquete.

Após a discussão de *drivers* de bloco, discutiremos outras classes de *driver*. O relógio é importante porque cada sistema tem um e porque é completamente diferente de todos os outros *drivers*. É também de interesse como uma exceção à regra de que todos dispositivos são de bloco ou de caractere, porque não se ajusta em nenhuma dessas categorias. Por fim, discutiremos o *driver* de terminal, que é importante em todos sistemas e, além disso, é um bom exemplo de *driver* de dispositivo de caractere.

Cada uma dessas seções descreve o hardware relevante, os princípios de software por trás do *driver*, uma visão geral da implementação e o código em si. Essa estrutura torna a leitura dessas seções útil mesmo para aqueles leitores que não estão interessados nos detalhes do código em si.

3.5.1 Visão Geral de *Drivers* de Dispositivo de Bloco no MINIX

Mencionamos anteriormente que os procedimentos principais de todas as tarefas de E/S têm uma estrutura similar. O MINIX sempre tem pelo menos três tarefas de dispositivo de bloco (o *driver* de disco de RAM, o *driver* de disquete e um entre vários *drivers* de disco rígido possíveis) compiladas no sistema. Além disso, uma tarefa de CD-ROM e um *driver* SCSI (*Small Computer Standard Interface*) pode ser compilado, no caso de suporte para tais dispositivos ser necessário. Embora o *driver* para cada um desses execute como um processo independente, o fato que todos eles são compilados como parte do executável do *kernel* torna possível compartilhar uma quantidade considerável de código, especialmente os procedimentos utilitários.

Cada *driver* de dispositivo de bloco precisa de alguma inicialização, naturalmente. O *driver* de disco de RAM precisa reservar alguma memória, o *driver* de disco rígido precisa determinar os parâmetros do hardware de disco rígido e assim por diante. Todos os *drivers* de disco são chamados individualmente para inicialização específica de hardware, mas depois de fazer o que possa ser necessário, cada *driver* chama a função que contém o laço principal comum, o qual é executado eternamente; não há nenhum retorno para o processo. Dentro do laço principal, uma mensagem é recebida, uma função para executar a operação necessária a cada mensagem é chamada e, então, uma mensagem de resposta é gerada.

O laço principal comum chamado por cada tarefa de *driver* de disco não é apenas uma cópia de uma função de biblioteca compilada em cada *driver*. Há somente uma cópia do código do laço principal no código binário do MINIX. A técnica utilizada é fazer cada um dos *drivers* individuais passar para o laço principal um parâmetro que consiste em um ponteiro para uma tabela dos endereços das funções que o *driver* utilizará para cada operação e, então, chamar essas funções indiretamente. Essa técnica também torna possível que os *drivers* compartilhem funções. A Figura 3-17 mostra um esboço do laço principal, de uma forma semelhante à da Figura 3-16. Declarações como

```
code = (*entry_points->dev_read)(&mess);
```

são chamadas indiretas de funções. Uma função *dev_read* diferente é chamada por cada *driver*, mesmo que cada um deles esteja executando o mesmo laço principal. Mas algumas outras operações, por exemplo *CLOSE*, são suficientemente simples para que mais de um dispositivo possa chamar a mesma função.

Essa utilização de uma única cópia do laço é uma boa ilustração do conceito de processo que introduzimos no Capítulo 1 e discutimos demoradamente no Capítulo 2. Há somente uma cópia do código executável na memória para o laço principal dos *drivers* de dispositivo de bloco, mas ela é executada como o laço principal de três ou de mais processos distintos. Cada um desses processos está provavelmente em um ponto diferente do código em um dado instante, e cada um está operando sobre seu próprio conjunto de dados e tem sua própria pilha.

Há seis possíveis operações que podem ser solicitadas para qualquer *driver* de dispositivo. Essas operações correspondem aos possíveis valores que podem ser localizados no campo *m.m_type* da mensagem na Figura 3-15. Elas são:

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. IOCTL
6. SCATTERED_IO

```
message mess;                                /* buffer de mensagem */

void shared_ io_task(struct driver_ table *entry_points) {
/* a inicialização é feita por cada tarefa antes de chamar esta */
  while (TRUE) {
       receive(ANY, &mess);
       caller = mess.source;
       switch(mess.type) {
           case READ:     rcode = (*entry_points->dev_read)(&mess); break;
           case WRITE:    rcode = (*entry_points->dev_write)(&mess);break;
           /* Outros casos entram aqui, incluindo OPEN, CLOSE e IOCTL */
           default:       rcode = ERROR;
       }
       mess.type = TASK_ REPLY;
       mess.status = rcode;                  /* código resultante */
       send(caller, &mess);
  }
}
```

Figura 3-17 Um procedimento principal compartilhado de tarefa de E/S utilizando chamadas indiretas.

A maioria dessas operações é provavelmente familiar para leitores com experiência em programação. No nível do *driver* de dispositivo, a maioria das operações está relacionada a chamadas de sistema com o mesmo nome. Por exemplo, os significados de READ e WRITE devem ser claros. Para cada uma dessas operações, um bloco de dados é transferido do dispositivo para a memória do processo que iniciou a chamada, ou vice-versa. Uma operação READ normalmente não resulta em um retorno para o processo até que a transferência de dados esteja completa, mas um sistema operacional pode *bufferizar* dados transferidos durante um WRITE para transferência real para o destino em um momento posterior e retornar para o processo imediatamente. Isso está ótimo no que diz respeito ao processo; ele, então, está livre para reutilizar o buffer de que o sistema operacional copiou os dados para gravar. *OPEN* e *CLOSE* para um dispositivo têm significados semelhantes para a maneira como as chamadas de sistema *OPEN* e *CLOSE* aplicam-se a operações em arquivos: uma operação *OPEN* deve verificar se o dispositivo está acessível ou retornar uma mensagem de erro se não, e uma operação *CLOSE* deve garantir que quaisquer dados *bufferizados* que foram gravados pelo processo foram completamente transferidos para seu destino final no dispositivo.

A operação *IOCTL* pode não ser tão familiar. Muitos dispositivos de E/S têm parâmetros operacionais que ocasionalmente devem ser examinados e talvez alterados. As operações *IOCTL* fazem isso. Um exemplo familiar é alterar a taxa de transmissão ou a paridade de uma linha de comunicações. Para dispositivos de bloco, as operações *IOCTL* são menos comuns. O exame ou a modificação do modo como um dispositivo de disco é particionado são feitos utilizando uma operação *IOCTL* no MINIX (embora isso pudesse ser feito igualmente bem, lendo e gravando um bloco de dados).

A operação *SCATTERED_IO* é, sem dúvida, a menos familiar dessas operações. Exceto com dispositivos de disco excessivamente rápidos (p. ex., o disco de RAM), é difícil obter um desempenho satisfatório de E/S de disco se todas as solicitações de disco forem para blocos individuais, um por vez. Uma solicitação *SCATTERED_IO* permite que o sistema de arquivos faça uma solicitação de leitura ou gravação de múltiplos blocos. No caso de uma operação READ, os blocos adicionais podem não ter sido solicitados pelo processo em cujo favor a chamada é feita; o sistema operacional tenta antecipar solicitações de dados futuras. Em tal solicitação nem todas as transferências solicitadas são necessariamente atendidas pelo *driver* de dispositivo. A solicitação para cada bloco pode ser modificada por um bit de sinalização que informa o *driver* de dispositivo de que a solicitação é opcional. De fato, o sistema de arquivos pode dizer, "seria ótimo ter todos esses dados, mas, realmente, não preciso de todos eles agora". O dispositivo pode fazer o que é melhor para ele. O *driver* de disquete, por exemplo, retornará todos os blocos de dados que ele pode ler de uma única trilha, efetivamente dizendo: "Darei esses a você, mas leva muito tempo para mover para outra trilha; solicite-me novamente o resto mais tarde."

Quando os dados devem ser gravados, não há nenhuma pergunta sobre se é opcional ou não gravar um bloco particular. Contudo, o sistema operacional pode *bufferizar* algumas solicitações de gravação na esperança de que a gravação de múltiplos blocos possa ser feita com maior eficiência do que tratar cada solicitação à medida que ela chega. Em uma solicitação *SCATTERED_IO*, seja para ler ou para gravar, a lista de blocos solicitada é classificada, e isso torna a operação mais eficiente do que tratar as solicitações aleatoriamente. Ademais, fazer uma única chamada para o *driver* para transferir múltiplos blocos reduz o número de mensagens enviadas dentro do MINIX.

3.5.2 Software Comum de *Driver* de Dispositivo de Bloco

As definições requeridas por todos os *drivers* de dispositivo de bloco estão localizadas em *driver.h*. A coisa mais importante neste arquivo é a estrutura *driver*, nas linhas

9010 a 9020, que é utilizada por cada *driver* para passar uma lista dos endereços das funções que ele utilizará para executar cada parte do seu trabalho. Também definida aqui está a estrutura *device* (linhas 9031 a 9034) que armazena as informações mais importantes sobre partições, o endereço de base e o tamanho, em unidades de byte. Esse formato foi escolhido para que nenhuma conversão fosse necessária ao trabalhar com dispositivos baseados em memória, maximizando a velocidade de resposta. Com discos reais há tantos outros fatores atrasando o acesso que a conversão para setores não é uma inconveniência significativa.

O laço principal e as funções compartilhadas de todas as tarefas de *driver* de bloco estão em *driver.c*. Depois de fazer qualquer inicialização específica de hardware que possa ser necessária, cada *driver* chama *driver_task*, passando uma estrutura *driver* como o argumento para a chamada. Depois de obter o endereço de um buffer utilizado para operações de DMA, o laço principal (linhas 9158 a 9199) é iniciado. Esse laço é executado eternamente; não há nenhum retorno para o processo.

O sistema de arquivos é o único processo do qual se supõe o envio de uma mensagem para uma tarefa de *driver*. O switch nas linhas 9165 a 9175 verifica isso. Uma interrupção de hardware remanescente é ignorada, qualquer outra mensagem maldirecionada resulta somente na impressão de um aviso na tela. Isso parece bastante inócuo, mas naturalmente o processo que enviou a mensagem errônea está, é provável, permanentemente bloqueado, esperando uma resposta. Em switch no laço principal, os primeiros três tipos de mensagem, *DEV_OPEN*, *DEV_CLOSE*, e *DEV_IOCTL*, resultam em chamadas indiretas utilizando endereços passados na estrutura *driver*. As mensagens *DEV_READ*, *DEV_WRITE*, e *SCATTERED_IO* resultam em chamadas diretas a *do_rdwt* ou *do_vrdwt*. Entretanto, a estrutura *driver* é passada como um argumento por todas as chamadas a partir de switch, sejam diretas ou indiretas, de maneira que todas as rotinas chamadas possam utilizá-la novamente mais tarde, conforme necessário.

Depois de fazer o que é solicitado na mensagem, algum tipo de limpeza pode ser necessária, dependendo da natureza do dispositivo. Para um disquete, por exemplo, isso pode envolver disparar um temporizador para desligar o motor da unidade de disco se outra solicitação não chegar logo. Uma chamada indireta também é utilizada para isso. Seguindo-se à limpeza, uma mensagem de resposta é criada e enviada ao processo (linhas 9194 a 9198).

A primeira coisa que cada tarefa faz depois de entrar no laço principal é chamar *init_buffer* (linha 9205), que atribui um buffer para utilização em operações de DMA. O mesmo buffer é utilizado por todas as tarefas de *driver*, se é que elas o utilizam — alguns *drivers* não utilizam DMA. As inicializações para cada entrada depois da primeira são redundantes, mas não causam nenhum mal. Seria mais incômodo codificar um teste para ver se a inicialização deve ser pulada.

O fato de essa inicialização ser desnecessária no final das contas deve-se a uma particularidade do hardware do IBM PC original, que requer que o buffer de DMA não ultrapasse um limite de 64K. Isto é, um buffer de DMA de 1K pode começar em 64510, mas não em 64514 porque um buffer que inicia no último endereço simplesmente se estende além do limite de 64K em 65536.

Essa regra irritante ocorre porque o IBM PC utilizou um chip de DMA antigo, o 8237A da Intel, que contém um contador de 16 bits. Um contador maior é necessário porque o DMA utiliza endereços absolutos, não endereços relativos, para um registrador de segmento. Em máquinas mais antigas que podem endereçar somente 1M de memória, os 16 bits de ordem baixa do endereço de DMA são carregados no 8237A e os 4 bits de ordem alta são carregados em um *latch** de 4 bits. Máquinas mais recentes utilizam um *latch* de 8 bits e podem endereçar 16M. Quando o 8237A vai de 0xFFFF a 0x0000, ele não atualiza o *latch*; então, o endereço de DMA repentinamente pula 64K para baixo na memória.

Um programa em C portável não pode especificar uma posição absoluta na memória para uma estrutura de dados, portanto, não há nenhuma maneira de evitar que o compilador coloque o buffer em uma posição não-utilizável. A solução é alocar uma matriz de bytes com o dobro do tamanho necessário em buffer (linha 9135) e reservar um ponteiro *tmp_buf* (linha 9136) a fim de realmente acessar essa matriz. *Init_buffer* faz uma configuração experimental de *tmp_buf* apontando para o início de buffer; então, testa para ver se isso permite espaço suficiente antes do limite de 64K ser alcançado. Se a configuração experimental não oferecer espaço suficiente, *tmp_buf* é incrementado pelo número de bytes realmente necessários. Assim algum espaço sempre é desperdiçado em uma das extremidades do espaço atribuído a buffer, mas nunca há uma falha decorrente do fato de o buffer cair no limite de 64K.

Computadores mais novos da família IBM PC têm melhores controladoras de DMA. Esse código poderia ser simplificado, e uma quantidade pequena de memória poderia ser requerida, se fosse possível assegurar que a máquina do usuário é imune a esse problema. Se você estiver considerando isso, entretanto, considere como se manifestaria o *bug* se você estivesse errado. Se um buffer de DMA de 1K é desejado, a chance é de 1 em 64 que haverá um problema em uma máquina com o chip de DMA antigo. Cada vez que o código-fonte do *kernel* for modificado de uma maneira que altera o tamanho do *kernel* compilado, há a mesma probabilidade de que o problema manifeste-se. Provavelmente, quando a falha ocorrer, no mês que vem ou no ano que vem, ela será atribuída à última modificação

*N. de T. Circuito ou elemento de circuito usado para manter um estado específico (como *on* ou *off*, verdadeiro lógico ou falso). O *latch* só muda de estado em resposta a uma entrada predeterminada. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

do código. "Recursos" inesperados de hardware como esse podem levar ao consumo de semanas procurando *bugs* excessivamente obscuros (e mais ainda quando, como esse, o manual técnico de referência não diz nenhuma palavra sobre eles).

Do_rdwt é a próxima função em *driver.c*. Essa, por sua vez, pode chamar três funções dependente de dispositivo apontadas pelos campos *dr_ prepare, dr_schedule* e *dr_finish* na estrutura *driver*. No que segue, utilizaremos a notação da linguagem de C **function_pointer* para indicar que estamos discutindo sobre a função apontada por *function_pointer*.

Depois de verificar se a contagem de byte na solicitação é positiva, *do_rdwt* chama **dr_prepare*. Isso deve ser bem-sucedido, uma vez que **dr_prepare* somente pode falhar se um dispositivo inválido é especificado em uma operação OPEN. Em seguida, uma estrutura-padrão *iorequest_s* (definida na linha 3194 em *include/minix/type.h*) é preenchida. Então, vem outra chamada indireta, desta vez para **dr_schedule*. Como veremos na discussão de hardware de disco na próxima seção, responder a solicitações de disco na ordem em que elas são recebidas pode ser ineficiente e essa rotina permite que um dispositivo particular trate solicitações da melhor maneira para o dispositivo. A indireção aqui mascara muitas possíveis variações na maneira como cada dispositivo atua. Para o disco de RAM, *dr_schedule* aponta para uma rotina que realmente executa a E/S, e a próxima chamada indireta, para *dr_finish*, é uma operação que não faz nada. Para um disco real, *dr_finish* aponta para uma rotina que leva a cabo todas as transferências de dados pendentes solicitadas em todas as chamadas anteriores para **dr_schedule* desde a última chamada a **dr_finish*. Como veremos, entretanto, em algumas circunstâncias, a chamada a **dr_finish* pode não resultar em uma transferência de todos os dados solicitados.

Em qualquer chamada que faz uma transferência de dados real, a contagem de *io_nbytes* na estrutura *iorequest_s* é modificada, retornando um número negativo se ocorreu um erro ou um número positivo que indica a diferença entre o número de bytes na solicitação original e o número transferido com êxito. Não é necessariamente um erro se nenhum byte for transferido; isso indica que o fim do dispositivo foi alcançado. Ao voltar para o laço principal, o código de erro negativo é retornado no campo *REP_STATUS* da mensagem de resposta se ocorreu um erro. Caso contrário, os bytes que restam ser transferidos são subtraídos da solicitação original no campo *COUNT* da mensagem (linha 9249) e o resultado (o número realmente transferido) é retornado no campo *REP_STATUS* da mensagem de resposta de *driver_task*.

A próxima função, *do_vrdwt*, manipula toda solicitação de E/S dispersa (*scattered*). Uma mensagem que requisita uma solicitação de E/S dispersa utiliza o campo *ADDRESS* para apontar para uma matriz de estruturas do tipo *iorequest_s*, cada uma especificando as informações necessárias para uma solicitação: o endereço do buffer, o deslocamento (*offset*) no dispositivo, o número de bytes e se a operação é uma leitura ou uma gravação. Todas as operações em uma solicitação serão para leitura ou para gravação e elas serão classificadas na ordem de blocos no dispositivo. Há mais trabalho a fazer do que a simples leitura ou a gravação realizada por *do_rdwt*, uma vez que a matriz de solicitações deve ser copiada para o espaço do *kernel*, mas uma vez que isso foi feito, as mesmas três chamadas indiretas para as rotinas dependentes de dispositivo **dr_prepare, *dr_schedule* e **dr_finish* são feitas. A diferença é que a chamada do meio, **dr_schedule*, é feita em um laço, uma vez para cada solicitação, ou até que um erro ocorra (linhas 9288 a 9290). Depois de terminar o laço, **dr_finish* é chamada uma vez e, então, a matriz de solicitações é copiada de volta para o lugar de onde veio. O campo *io_nbytes* de cada elemento na matriz terá sido alterado para refletir o número de bytes transferido e embora o total não seja copiado diretamente na mensagem de resposta que *driver_task* cria, o processo pode extrair o total dessa matriz.

Em uma solicitação de leitura de E/S dispersa, nem todas as transferências solicitadas na chamada a *dr_schedule* são necessariamente atendidas quando a chamada final a **dr_finish* é feita, como discutimos na seção anterior. O campo *io_request* na estrutura *iorequest_s* contém um bit de sinalização que informa ao *driver* de dispositivo se uma solicitação para esse bloco é opcional.

As próximas poucas rotinas em *driver.c* são para suporte geral das operações acima. Uma chamada **dr_name* pode ser utilizada para retornar o nome de um dispositivo. Para um dispositivo sem nenhum nome específico, a função *no_names* recupera o nome do dispositivo a partir da tabela de tarefas. Alguns dispositivos podem não requerer um serviço em particular, por exemplo, um disco de RAM não requer que qualquer coisa especial seja feita com uma solicitação *DEV_CLOSE*. A função *do_nop* cumpre esse papel aqui, retornando vários códigos que dependem do tipo de solicitação. As funções seguintes, *nop_finish* e *nop_cleanup*, são rotinas igualmente inócuas para dispositivos que não precisam dos serviços de **dr_finish* ou **dr_cleanup*.

Algumas funções de dispositivo de disco requerem um retardamento, por exemplo, para esperar o motor de uma unidade de disquetes começar a funcionar. Assim *driver.c* é um bom lugar para a próxima função, *clock_mess*, utilizada para enviar mensagens à tarefa de relógio. Ela é chamada com o número de tiques de relógio a esperar e o endereço de uma função a chamar quando o limite de tempo for atingido.

Por fim, *do_diocntl* (linha 9364) leva a cabo solicitações DEV_IOCTL para um dispositivo de bloco. É um erro se qualquer operação *DEV_IOCTL* que não ler *(DIOGETP)* ou gravar *(DIOSETP)* as informações de partição for solicitada. *Do_diocntl* chama a função **dr_prepare* do dispositivo para verificar se o mesmo é válido e obter um ponteiro para a estrutura de dispositivo que descreve a base e o tamanho da partição em unidades de byte. Em uma solicitação de leitura, ela chama a função **dr_geometry* do dis-

positivo para obter as últimas informações sobre a partição referentes a cilindros, a cabeçotes e a setores.

3.5.3 A Biblioteca de *Driver*

Os arquivos *drvlib.h* e *drvlib.e* contêm o código dependente de sistema que suporta partições de discos em computadores compatíveis com IBM PC.

O particionamento permite que um único dispositivo de armazenamento seja dividido em subdispositivos. Ele é mais comumente utilizado com discos rígidos, mas o MINIX também oferece suporte para particionar disquetes. Algumas razões para particionar um dispositivo de disco são:

1. A capacidade de disco é mais barata por unidade em discos grandes. Se dois ou mais sistemas operacionais com sistemas de arquivo diferentes são utilizados, é mais econômico particionar um único disco grande do que instalar múltiplos discos menores para cada sistema operacional.
2. Os sistemas operacionais podem ter limites para o tamanho de dispositivo que podem gerenciar. A versão do MINIX discutida aqui pode gerenciar um sistema de arquivos de 1 GB, mas versões mais antigas estão limitadas a 256MB. Qualquer espaço em disco além disso será desperdiçado.
3. Dois ou mais sistemas diferentes de arquivo podem ser utilizados por um sistema operacional. Por exemplo, um sistema de arquivos padrão pode ser utilizado para arquivos normais e um sistema de arquivos diferentemente estruturado pode ser utilizado para espaço de troca de memória virtual.
4. Pode ser conveniente pôr uma parte dos arquivos de um sistema em um dispositivo lógico separado. A colocação do sistema de arquivos raiz do MINIX em um dispositivo pequeno torna fácil fazer backup e facilita copiar ele para um disco de RAM em tempo de inicialização.

O suporte para partições de disco é específico da plataforma. Essa especificidade não está relacionada com o hardware. O suporte de partição é independente de dispositivo. Mas se mais de um sistema operacional rodará em um conjunto particular de hardware, todos devem concordar quanto ao formato para a tabela de partição. Em IBM PCs o padrão é dado pelo comando *fdisk* do MS-DOS, e outros sistemas operacionais, como MINIX, OS/2 e Linux, utilizam esse formato para que eles possam coexistir com o MS-DOS. Quando o MINIX é portado para outro tipo de máquina, faz sentido utilizar um formato de tabela de partição compatível com outros sistemas operacionais utilizados no novo hardware. Portanto, o código-fonte do MINIX para suportar partições em computadores IBM é configurado em *drvlib.c*, em vez de ser incluído em *drivers*, para tornar mais fácil portar o MINIX para um hardware diferente.

A estrutura de dados básica herdada dos projetistas de *firmware* é definida em *include/ibm/partition.h*, o qual é incluído por uma declaração #include em *drvlib.h*. Isso inclui as informações sobre a geometria cilindro–cabeçote–setor de cada partição, assim como os códigos para identificar o tipo de sistema de arquivos na partição e ativar um sinalizador que indica se é inicializável. A maioria dessas informações não é requerida pelo MINIX, uma vez que o sistema de arquivos tenha sido verificado.

A função *partition* (em *drvlib.c*, A linha 9521) é chamada quando um dispositivo de bloco é aberto pela primeira vez. Seus argumentos incluem uma estrutura *driver*, para que ela possa chamar funções específicas de dispositivo, um número inicial de dispositivo secundário e um parâmetro que indica se o estilo de particionamento é disquete, partição primária ou subpartição. Ela chama a função específica de dispositivo **dr_prepare* para verificar se o dispositivo é válido e para obter o endereço base e o tamanho em uma estrutura *device* do tipo mencionado na seção anterior. Então, ela chama *get_part_table* que determina se uma tabela de partição está presente e, se estiver, a lê. Se não houver nenhuma tabela de partição, o trabalho estará completo. Caso contrário, o número do dispositivo secundário da primeira partição é calculado, utilizando as regras para numeração de dispositivos secundários que se aplicam ao estilo de particionamento especificado na chamada original. No caso de partições primárias, a tabela de partição é classificada de tal modo que a ordem das partições de arquivo é consistente com a utilizada por outros sistemas operacionais.

Nesse ponto, outra chamada é feita para **dr_prepare*, desta vez utilizando o recém-calculado número de dispositivo da primeira partição. Se o subdispositivo é válido, então, um laço é feito sobre todas as entradas na tabela, verificando se os valores lidos da tabela no dispositivo não estão fora do intervalo obtido anteriormente para a base e para o tamanho do dispositivo inteiro. Se houver uma discrepância, a tabela na memória é ajustada para adaptar-se. Isso pode parecer paranóico, mas como as tabelas de partição podem ser gravadas por sistemas operacionais diferentes, um programador utilizando outro sistema poderia maliciosamente tentar utilizar a tabela de partição para algo inesperado ou poderia haver lixo na tabela em disco por alguma outra razão. Confiamos mais nos números que calculamos utilizando o MINIX. Melhor seguro que arrependido.

Ainda dentro do laço, para todas as partições no dispositivo, se a partição for identificada como uma partição MINIX, *partition* será recursivamente chamada para reunir as informações de subpartição. Se uma partição for identificada como uma partição estendida, a próxima função no arquivo, *extpartition*, será chamada em seu lugar.

Extpartition (linha 9593) realmente não tem nada a ver com o sistema operacional MINIX, portanto, não discutiremos seus detalhes. O MS-DOS utiliza partições estendidas, que são simplesmente outro mecanismo para criar subpartições. Para suportar comandos MINIX que podem ler e gravar arquivos MS-DOS, precisamos conhecer essas subpartições.

Get_part_table (linha 9642) chama *do_rdwt* para obter o setor em um dispositivo (ou subdispositivo) onde

uma tabela de partição está localizada. O argumento de deslocamento (*offset*) é zero se ela foi chamada para obter uma partição primária ou não-zero para uma subpartição. Ela verifica o número mágico (0xAA55) e retorna o status *true* ou *false* para indicar se uma tabela de partição válida foi localizada. Se uma tabela for localizada, ela a copia para o endereço de tabela que foi passado como um argumento.

Por fim, *sort* (linha 9676) classifica as entradas em uma tabela de partição pelo setor mais baixo. As entradas que são marcadas como não tendo nenhuma partição são excluídas da classificação, por isso aparecem no final, mesmo que possa ter um valor zero em seu campo de setor baixo. Essa classificação é um simples *buble sort**; não há nenhuma necessidade de utilizar um algoritmo extravagante para classificar uma lista de quatro itens.

3.6 DISCOS DE RAM

Agora veremos os *drivers* individuais de dispositivo de bloco e estudaremos vários deles detalhadamente. O primeiro que veremos é o *driver* de disco de RAM. Ele pode ser utilizado para oferecer acesso a qualquer parte da memória. Sua utilização principal é permitir que uma parte da memória seja reservada para utilização como um disco comum. Isso não oferece armazenamento permanente, mas uma vez que os arquivos tenham sido copiados para essa área, eles podem ser acessados com extrema rapidez.

Em um sistema como o MINIX, que foi projetado para trabalhar até em computadores com somente um disquete, o disco de RAM tem outra vantagem. Colocando o dispositivo raiz no disco de RAM, o disquete pode ser montado e desmontado à vontade, o que permite uma mídia removível. Se o dispositivo raiz fosse colocado no disquete, seria impossível salvar arquivos em disquetes, uma vez que o dispositivo raiz (o único disquete) não pode ser desmontado. Além disso, ter o dispositivo raiz no disco de RAM torna o sistema altamente flexível: qualquer combinação de disquetes ou de discos rígidos pode ser montada nele. Embora a maioria dos computadores de hoje em dia tenha discos rígidos, exceto os computadores utilizados em sistemas embutidos, o disco de RAM é útil durante a instalação, antes de o disco rígido estar pronto para utilização pelo MINIX, ou quando se quer utilizar o MINIX temporariamente sem fazer uma instalação completa.

*N. de T. Um algoritmo de pesquisa que se inicia no final de uma lista com n elementos e vai subindo aos poucos, testando o valor de cada par adjacente de elementos e trocando-os de posição, caso não estejam na ordem certa. O processo é repetido para os n - 1 elementos restantes até que a lista inteira esteja completamente classificada, com o maior valor posicionado no final da fila. O termo "classificação de bolhas" (*bubble sort*) origina-se do fato de que os elementos "mais leves" da lista (os menores elementos) vão subindo, como bolhas, até a superfície. Também chamado de *exchange sort* (classificação por troca). (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro, Editora Campus, 1998.)

3.6.1 Hardware e Software do Disco de RAM

A idéia por trás de um disco de RAM é simples. Um dispositivo de bloco é uma mídia de armazenamento com dois comandos: gravar um bloco e ler um bloco. Normalmente esses blocos são armazenados em memórias rotativas, como disquetes ou discos rígidos. Um disco de RAM é mais simples. Ele simplesmente utiliza uma porção pré-alocada da memória principal para armazenar os blocos. Um disco de RAM tem a vantagem de ter acesso instantâneo (nenhum retardo rotacional ou de busca), tornando-o conveniente para armazenar programas ou dados que são acessados com freqüência.

A propósito, vale a pena indicar brevemente uma diferença entre sistemas que suportam sistemas montados de arquivos e aqueles que não suportam (p. ex., o MS-DOS e o WINDOWS). Com sistemas montados de arquivos, o dispositivo raiz está sempre presente em uma posição fixa, e os sistemas de arquivos removíveis (i. e., discos) podem ser montados na árvore de arquivos para formar um sistema de arquivos integrado. Uma vez que tudo tenha sido montado, o usuário não precisa preocupar-se com saber em que dispositivo estão os arquivos.

Em contraposição, em sistemas tipo MS-DOS, o usuário deve especificar a posição de cada arquivo, seja explicitamente como em *B:\DIR\FILE* ou utilizando certos padrões (dispositivo atual, diretório atual, etc.). Com somente um ou dois disquetes, esse peso é gerenciável, mas, em um sistema de computador de grande porte, com dúzias de discos, precisar acompanhar dispositivos o tempo todo seria insuportável. Lembre-se de que o UNIX roda em sistemas que vão desde um IBM PC, passando por estações de trabalho e por supercomputadores até o Cray-2; o MS-DOS executa somente em sistemas pequenos.

A Figura 3-18 mostra a idéia por trás de um disco de RAM. O disco de RAM é dividido em *n* blocos, dependendo de quanta memória foi atribuída a ele. Cada bloco tem o mesmo tamanho que o tamanho de bloco utilizado em discos reais. Quando o *driver* recebe uma mensagem para ler ou para gravar um bloco, ele simplesmente calcula onde, na memória de disco de RAM os blocos solicitados estão e lê-os ou grava neles, em vez de ou em um disquete ou disco rígido. A transferência é feita chamando um procedimento de linguagem *assembly* que copia para o programa de usuário na velocidade máxima de que o hardware é capaz.

Um *driver* de disco de RAM pode suportar várias áreas da memória utilizadas como disco de RAM, cada uma distinguida por um número diferente de dispositivo secundário. Normalmente essas áreas são distintas, mas em algumas situações pode ser conveniente tê-las sobrepondo-se, como veremos na próxima seção.

Figura 3-18 Um disco de RAM.

3.6.2. Visão Geral do *Driver* de Disco de RAM no MINIX

O *driver* de disco de RAM é na realidade quatro *drivers* intimamente relacionados em um. Cada mensagem para ele especifica um dispositivo secundário como segue:

0: /dev/ram 1: /dev/mem 2: /dev/kmem
3: /dev/null

O primeiro arquivo especial listado acima, */dev/ram*, é um verdadeiro disco de RAM. Nem seu tamanho nem sua origem são definidos no *driver*. Eles são determinados pelo sistema de arquivos quando MINIX é inicializado. Por padrão, um disco de RAM do mesmo tamanho que o dispositivo de imagem do sistema de arquivos raiz é criado; então, o sistema de arquivos raiz pode ser copiado para ele. Um parâmetro de inicialização pode ser utilizado para especificar um disco de RAM maior que o sistema de arquivos raiz; ou, se a raiz não será copiada para a RAM, o tamanho especificado pode ser qualquer valor que se ajuste na memória e deixe memória suficiente para operação de sistema. Uma vez que o tamanho é conhecido, um bloco de memória suficientemente grande é localizado e removido do conjunto da memória, antes mesmo de o gerenciador de memória começar seu trabalho. Essa estratégia permite aumentar ou reduzir a quantidade de disco de RAM presente sem recompilar o sistema operacional.

Os dois próximos dispositivos secundários são utilizados para ler e para gravar memória física e memória do *kernel*, respectivamente. Quando */dev/mem* é aberto e lido, ele fornece o conteúdo de posições físicas da memória iniciando no endereço absoluto zero (os vetores de interrupções de modo real). Programas normais de usuário nunca fazem isso, mas um programa de sistema preocupado com depurar o sistema talvez necessite dessa facilidade. A operação de abrir */dev/mem* e gravar nele mudará os vetores de interrupção. É desnecessário dizer, isso somente deve ser feito com a maior cautela por um usuário experimentado que saiba exatamente o que está fazendo.

O arquivo especial */dev/kmem* é como */dev/mem*, exceto que o byte 0 desse arquivo é o byte 0 da memória de dados do *kernel*, uma posição cujo endereço absoluto varia, dependendo do tamanho do código do *kernel* do MINIX. Ele também é utilizado principalmente para depuração e programas muito especiais. Note que as áreas de disco de RAM cobertas por esses dois dispositivos secundários sobrepõem-se. Se você souber exatamente como o *kernel* está colocado na memória, você pode abrir */dev/mem*, procurar o começo da área de dados do *kernel* e ver exatamente a mesma coisa que é lida desde o início de */dev/kmem*. Mas, se recompilar o *kernel*, alterar seu tamanho ou se em uma versão subseqüente do MINIX o *kernel* for movido para outro lugar na memória, você precisará procurar uma quantidade diferente em */dev/mem* para ver a mesma coisa que você vê no início de */dev/kmem*. Ambos esses arquivos especiais deverão ser protegidos de modo que ninguém, exceto o superusuário, possa utilizá-los.

O último arquivo nesse grupo, */dev/null*, é um arquivo especial que recebe dados e joga-os fora. Ele é comumente utilizado em comandos de *shell* quando o programa que está sendo chamado gera saída que não é necessária. Por exemplo,

```
a.out >/dev/null
```

executa o programa *a.out* mas descarta sua saída. O *driver* de disco de RAM efetivamente trata esse dispositivo secundário como se ele tivesse tamanho zero, assim nenhum dado jamais é copiado para ele ou a partir dele.

O código para tratar */dev/ram*, */dev/mem* e */dev/kmem* é idêntico. A única diferença *entre* eles é que cada um corresponde a uma porção diferente de memória, indicada pelas matrizes *ram_origin* e *ram_limit*, cada uma indexada pelo número de dispositivo secundário.

3.6.3 Implementação do *Driver* de Disco de RAM no MINIX

Como com outros *drivers* de disco, o laço principal do disco de RAM está no arquivo *driver.c*. O suporte específico de dispositivo para dispositivos de memória está em *memory.c*. A matriz *m_geom* (linha 9721) guarda a base e o tamanho de cada um dos quatro dispositivos de memória. A estrutura *m_dtab* de *driver* nas linhas 9733 a 9743 define as chamadas de dispositivo de memória que serão feitas a partir do laço principal. Quatro das entradas nessa tabela são rotinas que pouco ou nada fazem em *driver.c*, um indício seguro de que a operação de um disco de RAM não é terrivelmente complicada. O procedimento principal *mem_task* (linha 9749) chama uma função para fazer alguma inicialização local. Depois disso, ela chama o laço principal, que recebe mensagens, despacha para o procedimento apropriado e envia as respostas. Não há nenhum retorno para *mem_task* na conclusão.

Em uma operação de leitura ou de gravação o laço principal faz três chamadas: uma para preparar o dispositivo, uma para agendar as operações de E/S e uma para terminar a operação. Para um dispositivo de memória uma chamada para *m_prepare* é a primeira dessas. Ela verifica se um dispositivo secundário válido foi solicitado e, então, retorna o endereço da estrutura que armazena o endereço de base e o tamanho da área de RAM solicitada. A segunda chamada é para *m_schedule* (linha 9774). Essa faz todo o trabalho. Para dispositivos de memória, o nome dessa função é equivocado: por definição, qualquer posição é tão acessível quanto qualquer outra na memória de acesso aleatório e, assim, não há nenhuma necessidade de fazer qualquer agendamento, como há no caso de um disco que tem um braço móvel.

A operação do disco de RAM é tão simples e rápida que nunca há qualquer razão para adiar uma solicitação, e a primeira coisa feita por essa função é limpar o bit que pode ser ativado por uma chamada de E/S dispersa para indicar que conclusão de uma operação é opcional. O endereço de destino passado na mensagem aponta para uma posição no espaço de memória do processo, e o código nas linhas 9792 a 9794 converte isso em um endereço absoluto na memória de sistema e, então, verifica se é um endereço válido. A transferência de fato dos dados acontece na linha 9818 ou na linha 9820 e é uma simples operação de cópia dos dados de um lugar para outro.

Um dispositivo de memória não precisa de um terceiro passo para finalizar uma operação de leitura ou de gravação, e a entrada correspondente em *m_dtab* é uma chamada para *nop_finish*.

A abertura de um dispositivo de memória é feita por *m_do_open* (linha 9829). O trabalho principal é feito chamando *m_prepare* para verificar se um dispositivo válido está sendo referenciado. No caso de uma referência a */dev/mem* ou */dev/kmem*, uma chamada a *enable.iop* (no arquivo *protect.c*) é feita para mudar o nível atual de privilégio da CPU. Isso não é necessário para acessar memória. É um truque para lidar com outro problema. Lembre-se de que a classe de CPUs Pentium implementa quatro níveis de privilégio. Os programas de usuário estão no nível menos privilegiado. Os processadores da Intel também têm um recurso arquitetônico que não está presente em muitos outros sistemas: um conjunto separado de instruções para endereçar portas de E/S. Nesses processadores as portas de E/S são tratadas separadamente da memória. Normalmente, uma tentativa por parte de um processo de usuário de executar uma instrução que endereça uma porta de E/S causa uma falha geral de proteção. Entretanto, há razões válidas para o MINIX permitir que os usuários escrevam programas que possam acessar portas, especialmente em sistemas pequenos. Assim *enable_iop* muda os bits de nível de proteção de E/S (I/O *Protection Level* — IOPL) da CPU para permitir isso. O efeito é dar a um processo que tem permissão para abrir */dev/mem* ou / *dev/kmem* o privilégio adicional de acessar portas de E/S. Em uma arquitetura em que os dispositivos de E/S são endereçados como posições da memória, os bits *rwx* para esses dispositivos automaticamente cobrem o acesso para E/S. Se esse recurso fosse oculto, talvez fosse considerado uma falha de segurança, mas agora você sabe disso. Se planeja utilizar o MINIX para controlar o sistema de segurança de um banco, talvez você queira recompilar o *kernel* sem essa função.

A próxima função, *m_init* (linha 9849) é chamada somente uma vez, quando *mem_task* é chamada pela primeira vez. Ela define o endereço de base e o tamanho de / *dev/kmem* e também define o tamanho de */dev/mem* como 1MB, 16MB ou 4 GB–1, dependendo se o MINIX está sendo executado no modo 8088, 80286 ou 80386. Esses são os tamanhos máximos suportados pelo MINIX e não tem nada a ver com a quantidade de RAM instalada na máquina.

O disco de RAM suporta várias operações IOCTL em *m_ioctl* (linha 9874). *MIOCRAMSIZE* é uma maneira conveniente de o sistema de arquivos configurar o tamanho do disco de RAM. A operação *MIOCSPSINFO* é utilizada tanto pelo sistema de arquivos como pelo gerenciador de memória para configurar os endereços de suas partes da tabela de processos na tabela *psinfo*, onde o programa utilitário *ps* pode recuperá-los utilizando uma operação *MIOCGPSINFO*. *Ps* é um programa UNIX padrão cuja implementação é complicada pela estrutura de *microkernel* do MINIX, que coloca as informações da tabela de processos requeridas pelo programa em vários lugares diferentes. A chamada de sistema IOCTL é uma maneira conveniente de tratar esse problema. Caso contrário, uma nova versão de *ps* teria de ser compilada cada vez que uma nova versão do MINIX fosse compilada.

A última função em *memory.c.* é *m_geometry* (linha 9934). Os dispositivos de memória não têm uma geometria de cilindros, trilhas e setores por trilha como unidades de disco mecânicas, mas no caso de o disco de RAM ser solicitado, ele atenderá fingindo ter.

3.7 DISCOS

O disco de RAM é uma boa apresentação para *drivers* de disco (porque é bem simples), mas os discos reais apresentam diversas questões que ainda não abordamos. Nas próximas seções primeiro diremos algumas palavras sobre hardware de disco e, então, daremos uma olhada em *drivers* de disco em geral, e o *driver* de disco rígido do MINIX em particular. Não examinaremos o *driver* de disquete detalhadamente, mas repassaremos alguns dos pontos em que um *driver* de disquete difere de um *driver* de disco rígido.

3.7.1 Hardware de Disco

Todos os discos reais são organizados em cilindros, cada um contendo um número de trilhas igual ao número de cabeçotes empilhados verticalmente. As trilhas são divididas em setores, e o número de setores ao redor da circunferência é geralmente de 8 a 32 em disquetes e até várias centenas em alguns discos rígidos. Os projetos mais simples têm o mesmo número de setores em cada trilha. Todos os setores contêm o mesmo número de bytes, embora basta pensar um pouco para perceber que os setores perto da borda exterior do disco serão fisicamente mais longos que aqueles perto do eixo. Mas o tempo de leitura ou de gravação de cada setor é o mesmo. A densidade de dados é obviamente mais alta nos cilindros mais internos, e alguns projetos de disco requerem uma mudança na corrente da unidade para os cabeçotes de leitura-gravação para as trilhas interiores. Isso é tratado pelo hardware da controladora de disco e não é visível para o usuário (nem pelo implementador de um sistema operacional).

A diferença na densidade de dados entre trilhas exteriores e interiores significa um sacrifício em termos de capacidade, e existem sistemas mais sofisticados. Foram feitas experiências com projetos de disquete que giram em velocidades mais altas quando os cabeçotes estão sobre as trilhas exteriores, o que permite mais setores nessas trilhas, aumentando a capacidade do disco. Entretanto, esse tipo de disco não é suportado por nenhum sistema para o qual MINIX está disponível atualmente. Os modernos discos rígidos disponíveis atualmente também têm mais setores por trilha em trilhas exteriores que em trilhas interiores. Esses discos são as unidades **IDE (*Integrated Device Electronics*)** e o processamento sofisticado feito pela eletrônica integrada na unidade oculta os detalhes. Para o sistema operacional, eles parecem ter uma geometria simples com o mesmo número de setores em cada trilha.

A eletrônica da unidade e da controladora é tão importante quanto o hardware mecânico. O elemento principal da placa controladora que é conectado no *backplane** do computador é um circuito integrado especializado, realmente um pequeno microcomputador. Para um disco rígido, os circuitos da placa controladora podem ser mais simples que um disquete, mas isso é porque o próprio disco rígido tem uma poderosa controladora eletrônica interna. Um recurso de dispositivo que tem implicações importantes para o *driver* de disco é a possibilidade de a controladora fazer busca em duas ou mais unidades ao mesmo tempo. Isso é conhecido como **busca sobreposta** (*overlapped seeks*). Enquanto a controladora e o software estão esperando uma busca ser completada em uma unidade, a controladora pode iniciar uma busca em outra unidade. Muitas controladoras também podem ler ou gravar em uma unidade enquanto buscam em uma ou mais outras unidades, mas uma controladora de disquete não pode ler ou gravar em duas unidades ao mesmo tempo. (Leitura ou gravação requerem que a controladora mova bits em uma escala de tempo de microssegundos, então, uma transferência consome a maior parte do seu poder de computação.) A situação é diferente para discos rígidos com controladoras integradas, e em um sistema com mais de um desses discos rígidos eles podem operar simultaneamente, pelo menos no que tange à transferência entre o disco e a memória de buffer da controladora. Entretanto, é possível somente uma transferência por vez entre a controladora e a memória do sistema. A capacidade de executar duas ou mais operações ao mesmo tempo pode reduzir o tempo de acesso médio consideravelmente.

A Figura 3-19 compara parâmetros de disquetes de dupla densidade e de duas faces, a mídia de armazenamento padrão para o IBM PC original, com parâmetros de um típico disco rígido de média capacidade, como os que podem ser encontrados em um computador baseado em Pentium. O MINIX utiliza blocos de 1K, assim com qualquer um desses formatos de disco os blocos utilizados pelo software consistem em dois setores consecutivos, que sempre são lidos ou gravados juntos como uma unidade.

Algo de que se deve estar ciente ao analisar as especificações dos discos rígidos modernos é que a geometria especificada e utilizada pelo software de *driver*, pode ser diferente do formato físico. O disco rígido descrito na Figura 3-19, por exemplo, é especificado com "parâmetros recomendados de configuração" de 1048 cilindros, 16 cabeçotes e 63 setores por trilha. A eletrônica da controladora montada no disco converte os parâmetros lógicos de cabeçote e de setor fornecidos pelo sistema operacional nos correspondentes físicos utilizados pelo disco. Esse é outro exemplo de um projeto comprometido em manter a compatibilidade com sistemas mais antigos, nesse caso *firmware* antigo. Os projetistas do IBM PC original reservaram somente um campo de 6 bits para a contagem de setores da ROM BIOS e um disco que tenha mais de 63 setores físicos por trilha deve trabalhar com um conjunto artificial de parâmetros lógicos de discos. Nesse caso, as especificações do fabricante afirmam que há realmente quatro cabeçotes e assim pareceria que há realmente 252 setores por trilha, como indicado na figura. Isso é uma supersimplificação, porque discos como esses têm mais setores nas trilhas mais exter-

*N. de T. Uma placa ou estrutura de circuitos que suporta outras placas de circuitos, de dispositivos e as interconexões entre os dispositivos e fornece força e sinais de dados aos dispositivos suportados. (*Dicionário de Informática*. Microsoft Press. Rio de Janeiro. Editora Campus. 1998.)

nas do que nas trilhas internas. O disco descrito na figura tem quatro cabeçotes físicos, mas na realidade há ligeiramente mais de 3.000 cilindros. Os cilindros são agrupados em uma dúzia de zonas que têm 57 setores por trilha nas zonas mais internas e até 105 cilindros por trilha nos cilindros mais externos. Esses números não serão encontrados nas especificações do disco, e as traduções feitas pela eletrônica da unidade tornam desnecessário para nós conhecer esses detalhes.

3.7.2 Software de Disco

Nesta seção, veremos algumas questões relacionadas a *drivers* de disco em geral. Primeiro, considere quanto tempo leva para ler ou para gravar um bloco de disco. O tempo necessário é determinado por três fatores:

1. O tempo de busca (o tempo de mover o braço para o cilindro adequado).
2. O retardo rotacional (o tempo para o setor adequado girar sob o cabeçote).
3. Tempo real de transferência de dados.

Para a maioria dos discos, o tempo de busca domina os outros dois tempos, portanto, reduzir tempo de busca médio significa melhorar o desempenho do sistema substancialmente.

Os dispositivos de disco são propensos a erros. Algum tipo de verificação de erros, uma soma de verificação ou uma verificação de redundância cíclica, sempre é gravado junto com os dados em cada setor em um disco. Mesmo os endereços de setores gravados quando o disco é formatado têm dados de verificação. O hardware da controladora de disquete pode informar quando um erro é detectado, mas o software, então, deve decidir o que fazer com ele. As controladoras de disco rígido freqüentemente arcam com a maior parte desse peso.

Particularmente com discos rígidos, o tempo de transferência para setores consecutivos dentro de uma trilha pode ser muito rápido. Assim, ler mais dados do que é solicitado e fazer *cache* deles na memória pode ser muito eficiente para acelerar o acesso de disco.

Algoritmos de Agendamento do Braço de Disco

Se o *driver* de disco aceita uma solicitação por vez e executa-as nessa ordem, isto é, primeiro a entrar, primeiro a ser servido (*First-Come, First-Served* — FCFS), pouco pode ser feito para otimizar o tempo de busca. Entretanto, outra estratégia é possível quando o disco é muito carregado. É possível que enquanto o braço esteja fazendo uma busca em favor de uma solicitação, outras solicitações de disco possam ser geradas por outros processos. Muitos *drivers* de disco mantêm uma tabela, indexada por número de cilindro, com todas as solicitações pendentes para cada cilindro encadeadas juntas em uma lista encadeada encabeçada pelas entradas da tabela.

Dado esse tipo de estrutura de dados, podemos melhorar o algoritmo de agendamento primeiro a entrar, primeiro a ser servido. Para ver como, considere um disco com 40 cilindros. Chega uma solicitação para ler um bloco no cilindro 11. Enquanto a busca do cilindro 11 está em progresso, chega uma nova solicitação para os cilindros 1, 36, 16, 34, 9 e 12, nessa ordem. Eles são inseridos na tabela de solicitações pendentes, com uma lista encadeada separada para cada cilindro. As solicitações são mostradas na Figura 3-20.

Parâmetro	Disquete IBM de 360KB	Disco rígido WD de 540MB
Número de cilindros	40	1048
Trilhas por cilindro	2	4
Setores por trilha	9	252
Setores por disco	720	1056384
Bytes por setor	512	512
Bytes por disco	368640	540868608
Tempo de busca (cilindros adjacentes)	6ms	4ms
Tempo de busca (caso médio)	77ms	11ms
Tempo de rotação	200ms	13ms
Tempo de parada/partida do motor	250ms	9s
Tempo para transferir 1 setor	22ms	53µs

Figura 3-19 Os parâmetros de disco para o disquete IBM PC original de 360KB e um disco rígido Western Digital WD AC2540 de 540MB.

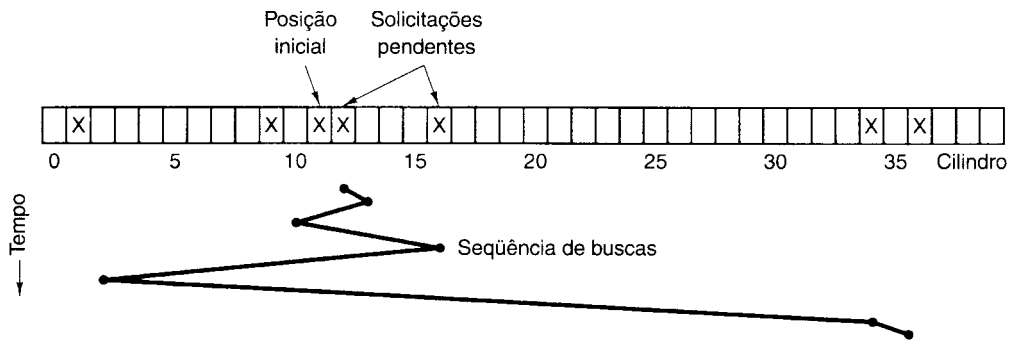



Figura 3-20 Algoritmo de agendamento de disco busca mais curta primeiro (*Shortest Seek First* – SSF).

Quando a solicitação atual (para o cilindro 11) termina, o *driver* de disco pode escolher qual solicitação ele deve tratar em seguida. Utilizando FCFS, ele iria em seguida para o cilindro 1, depois para o 36 e assim por diante. Esse algoritmo exigiria movimentos de braço de 10, 35, 20, 18, 25 e 3, respectivamente, em um total de 111 cilindros.

Alternativamente, ele sempre poderia tratar a solicitação mais próxima a seguir, para minimizar o tempo de busca. Dadas as solicitações na Figura 3-20, a seqüência é 12, 9, 16, 1, 34 e 36, mostrada como uma linha serrilhada na parte inferior da Figura 3.20. Com essa seqüência, os movimentos de braço são 1, 3. 7, 15. 33 e 2, em um total de 61 cilindros. Esse algoritmo, **busca mais curta primeiro** (*Shortest Seek First* – SSF), reduz o movimento total de braço em quase metade se comparado com o FCFS.

Infelizmente, o SSF tem um problema. Suponha que mais solicitações continuem chegando enquanto as solicitações da Figura 3-2 0 estão sendo processadas. Por exemplo, se, depois de ir para o cilindro 16, uma nova solicitação para o cilindro 8 for feita, essa solicitação terá prioridade sobre o cilindro 1. Se então chegar uma solicitação para o cilindro 13, o braço irá em seguida para 13 em vez de 1. Com um disco muito carregado, o braço tenderá a permanecer no meio do disco a maior parte do tempo, portanto, as solicitações em qualquer extremo terão de esperar até que uma flutuação estatística na carga faça com que não haja nenhuma solicitação perto do meio. As solicitações longe do meio podem obter um serviço deficiente. As metas de tempo mínimo de resposta e de imparcialidade estão em conflito aqui.

Edifícios altos também precisam lidar com esse tipo de compensação. O problema de agendar um elevador em um edifício alto é semelhante ao do agendamento de um braço de disco. As solicitações entram continuamente chamando o elevador para andares (cilindros) aleatórios. O microprocessador que controla o elevador poderia facilmente acompanhar a seqüência em que os clientes apertam o botão de chamada e serviria-os, utilizando FCFS. Ele também poderia utilizar SSF.

Entretanto, a maioria dos elevadores utiliza um algoritmo diferente para atender às exigências contraditórias de eficiência e de imparcialidade. Eles continuam a mover-se na mesma direção até que não haja mais solicitações destacadas nessa direção; então, eles mudam de direção. Esse algoritmo, conhecido tanto no mundo da computação como no mundo dos elevadores como **algoritmo do elevador,** requer que o software mantenha 1 bit: o bit de direção atual, UP ou DOWN. Quando uma solicitação termina, o *driver* de disco ou de elevador verifica o bit. Se é UP, o braço ou a cabine é movido para a próxima solicitação acima. Se nenhuma solicitação está pendente em posições mais altas, o bit de direção é invertido. Quando o bit é configurado como DOWN, o movimento é para a próxima solicitação abaixo, se houver alguma.

A Figura 3-21 mostra o algoritmo do elevador utilizando as mesmas sete solicitações da Figura 3-20, supondo que o bit de direção era inicialmente UP. A ordem em que os cilindros são servidos é 12, 16, 34, 36, 9 e 1, o que resulta em movimentos de braço de 1, 4, 18, 2, 27 e 8, para um total de 60 cilindros. Nesse caso, o algoritmo do elevador é ligeiramente melhor do que o SSF, embora seja geralmente pior. Uma propriedade interessante que o algoritmo do elevador tem é que dada qualquer coleção de solicitações, o limite superior para o movimento total é fixo: ele é simplesmente duas vezes o número de cilindros.

Uma ligeira modificação desse algoritmo que tem uma variância menor nos tempos de resposta (Teory, 1972) é avançar sempre na mesma direção. Quando o cilindro com a numeração mais alta com uma solicitação pendente é servido, o braço vai para o cilindro com a numeração mais baixa com uma solicitação pendente e, então, continua a mover-se para cima. De fato, o cilindro com numeração mais baixa é simplesmente considerado como estando acima do cilindro com numeração mais alta.

Algumas controladoras de disco oferecem um modo de o software inspecionar o número do setor atual sob o cabeçote. Com uma controladora desse tipo, outra otimização é possível. Se duas ou mais solicitações para o mesmo cilin-

dro estiverem pendentes, o *driver* pode fazer uma solicitação para o setor que passará sob o cabeçote em seguida. Note que quando múltiplas trilhas estão presentes em um cilindro, podem ser feitas solicitações consecutivas para trilhas diferentes sem nenhuma penalidade. A controladora pode selecionar qualquer dos seus cabeçotes instantaneamente, porque a seleção de cabeçote não envolve nenhum movimento de braço nem retardo rotacional.

Com um disco rígido moderno, a taxa de transferência de dados é tão mais rápida que a de um disquete que algum tipo de *cache* automático é necessário. Geralmente qualquer solicitação de leitura de um setor faz com que esse setor e todo o resto da trilha atual seja lido, dependendo de quanto espaço estiver disponível na memória de *cache* da controladora. O disco de 540M descrito na Figura 3-19 tem um *cache* de 64K ou de 128K. A utilização do *cache* é determinada dinamicamente pela controladora. Em seu modo mais simples, o *cache* é dividido em duas seções, uma para ler e uma para gravar

Quando várias unidades estão presentes, uma tabela de solicitações pendentes deve ser mantida para cada unidade separadamente. Sempre que qualquer unidade está desocupada, deve ser feita uma busca para mover o seu braço para o cilindro onde será necessário em seguida (supondo que a controladora permite busca sobreposta). Quando a transferência atual terminar, uma verificação pode ser feita para ver se qualquer unidade está posicionada no cilindro correto. Se uma ou mais estiverem, a próxima transferência poderá ser iniciada em uma unidade que já está no cilindro correto. Se nenhum braço estiver no lugar certo, o *driver* deve fazer uma nova busca na unidade que acabou de completar uma transferência e esperar a interrupção seguinte para ver qual braço chega ao seu destino primeiro.

Tratamento de Erros

Os discos de RAM não precisam preocupar-se com otimização rotacional ou de busca: em qualquer instante todos os blocos podem ser lidos ou gravados sem qualquer movimento físico. Outra área em que os discos de RAM são mais simples que os discos reais é o tratamento de erros. Os discos de RAM sempre funcionam: os reais nem sempre funcionam. Eles estão sujeitos a uma ampla variedade de erros. Alguns dos mais comuns são:

1. Erro de programação (p. ex., solicitação para um setor inexistente).
2. Erro transitório de soma de verificação (p. ex., causado por pó no cabeçote).
3. Erro permanente de soma de verificação (p. ex., bloco de disco fisicamente danificado).
4. Erro de busca (p. ex., o cilindro foi enviado para o braço 6 mas acabou indo para o 7).
5. Erro de controladora (p. ex., a controladora recusa-se a aceitar comandos).

Cabe ao *driver* de disco tratar cada um desses erros da melhor maneira que puder.

Os erros de programação ocorrem quando o *driver* instrui a controladora para buscar um cilindro inexistente, ler de um setor inexistente, utilizar um cabeçote inexistente ou transferir para ou de uma memória inexistente. A maioria das controladoras verifica os parâmetros fornecidos e queixa-se se eles forem inválidos. Em teoria, esses erros nunca devem ocorrer, mas o que o *driver* deve fazer se a controladora indicar que aconteceu? Para um sistema doméstico, a melhor coisa a fazer é parar e imprimir uma mensagem como "Chame o programador" para que o erro possa ser verificado e corrigido. Para um produto comercial de software em utilização em milhares de pontos ao redor do mundo, essa abordagem é menos atraente. Provavelmente a única coisa a fazer é encerrar a solicitação atual ao disco com um erro e esperar que ela não se repita com muita freqüência.

Erros transitórios de soma de verificação são causados por partículas de pó no ar que acabam ficando entre o cabeçote e a superfície de disco. A maioria das vezes, esses erros podem ser eliminados simplesmente repetindo-se a operação algumas vezes. Se o erro persistir, o bloco precisa ser marcado como um **bloco defeituoso** e evitado.

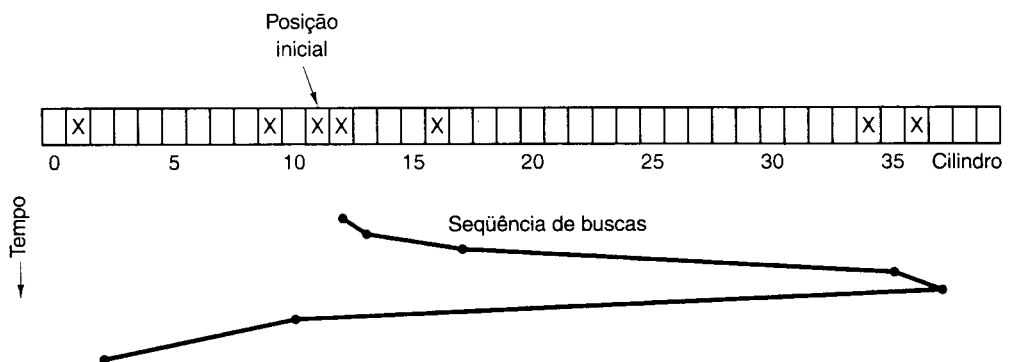



Figura 3-21 O algoritmo do elevador para agendamento de solicitações de disco.

Uma maneira de evitar blocos defeituosos é escrever um programa muito especial que pega uma lista de blocos defeituosos como entrada e cuidadosamente cria um arquivo que contém todos os blocos defeituosos. Uma vez que o arquivo tenha sido criado, o alocador de disco pensará que os blocos estão ocupados e nunca os atribuirá. Contanto que ninguém jamais tente ler o arquivo de blocos defeituosos, nenhum problema ocorrerá.

Não ler o arquivo bloco defeituoso é mais fácil dizer do que fazer. Muitos discos são salvos em backup copiando-se seu conteúdo uma trilha por vez para uma unidade de disco ou fita de backup. Se esse procedimento for seguido, os blocos defeituosos causarão problema. Fazer backup do disco um arquivo por vez é mais lento mas resolverá o problema, desde que o programa de backup saiba o nome do arquivo de blocos defeituosos e evite copiá-lo.

Outro problema que não pode ser resolvido com um arquivo blocos defeituosos é um bloco defeituoso em uma estrutura de dados do sistema de arquivos que deve estar em uma posição fixa. Quase todos os sistemas de arquivos têm pelo menos uma estrutura de dados cuja posição é fixa, assim ela pode ser localizada facilmente. Em um sistema de arquivos particionado pode ser possível reparticionar e contornar uma trilha defeituosa, mas um erro permanente nos primeiros setores de qualquer disquete ou disco rígido geralmente significa que o disco não é utilizável.

Controladoras "inteligentes" reservam algumas trilhas normalmente não-disponíveis para programas de usuário. Quando uma unidade de disco é formatada, a controladora determina quais blocos são defeituosos e automaticamente substitui a trilha defeituosa por uma das trilhas de reserva. A tabela que mapeia trilhas defeituosas para trilhas de reserva é mantida na memória interna da controladora e no disco. Essa substituição é transparente (invisível) para o *driver*, exceto que seu algoritmo do elevador cuidadosamente elaborado pode ter um pobre desempenho se a controladora secretamente estiver utilizando o cilindro 800 sempre que cilindro 3 for solicitado. Hoje, a tecnologia de fabricar superfícies de gravação de disco é melhor do que era antigamente, mas ainda não é perfeita. Entretanto, a tecnologia de esconder as imperfeições do usuário também melhorou. Em discos rígidos como o descrito na Figura 3-19, a controladora também gerencia novos erros que podem desenvolver-se com a utilização, atribuindo blocos substitutos permanentemente quando determina que um erro é irrecuperável. Com esses discos, o software de *driver* raramente vê qualquer indicação de que há quaisquer blocos defeituosos.

Erros de busca são causados por problemas mecânicos no braço. A controladora monitora a posição do braço internamente. Para realizar uma busca, ela emite uma série de pulsos para o motor do braço, um pulso por cilindro, para mover o braço para o novo cilindro. Quando o braço chega ao seu destino, a controladora lê o número real do cilindro (gravado quando a unidade foi formatada). Se o braço estiver no lugar errado, um erro de busca ocorreu.

A maioria das controladoras de disco rígido corrige problemas de busca automaticamente, mas muitas controladoras de disquete (IBM-PCs inclusive) simplesmente configuram um bit de erro e deixam o resto para o *driver*, o qual trata esse erro dando um comando RECALIBRATE, para mover o braço o máximo possível para fora e redefinir a idéia interna que a controladora tem sobre o cilindro atual para 0. Normalmente isso resolve o problema. Se não, a unidade deve ser reparada.

Como vimos, a controladora é realmente um pequeno computador especializado, completa com software, variáveis, buffers e ocasionalmente, *bugs*. Às vezes, uma seqüência única de eventos como uma interrupção em uma unidade ocorrendo simultaneamente com um comando RECALIBRATE para outra unidade desencadeará um erro e fará com que a controladora entre em um laço ou perca a trilha do que estava fazendo. Projetistas de controladoras normalmente planejam para o pior e oferecem um pino no chip que, quando conectado, força a controladora a esquecer o que estava fazendo e reinicializar-se. Se tudo mais falhar, o *driver* de disco pode configurar um bit para invocar esse sinal e chamar a controladora. Se isso não ajudar, tudo o que o *driver* pode fazer é imprimir uma mensagem e desistir.

Fazendo Cache *de uma Trilha por Vez*

O tempo necessário para buscar um novo cilindro é normalmente muito maior do que o retardo rotacional e sempre muito maior do que o tempo de transferência. Em outras palavras, uma vez que o *driver* caiu no problema de mover o braço para algum lugar, dificilmente faz diferença se lê um setor ou uma trilha inteira. Esse efeito é especialmente verdadeiro se a controladora oferece sensibilidade rotacional, então, o *driver* pode ver qual setor está atualmente sob o cabeçote e fazer uma solicitação para o próximo setor, tornando possível assim ler uma trilha por tempo de rotação. (Normalmente leva o tempo de meia rotação mais um setor somente para ler-se um único setor, em média.)

Alguns *drivers* de disco tiram proveito dessa propriedade mantendo um *cache* secreto de uma trilha por vez, desconhecido pelo software independente de dispositivo. Se uma seção que está no *cache* for solicitada, nenhuma transferência de disco é necessária. Uma desvantagem de fazer *cache* uma trilha por vez (além da complexidade do software e do espaço de buffer necessários) é que as transferências do *cache* para o programa que fez chamada terão de ser feitas pela CPU utilizando um laço programado, em vez de deixar o hardware de DMA fazer o trabalho.

Algumas controladoras levam esse processo um passo adiante e fazem um *cache* de uma trilha por vez na própria memória interna, transparente para o *driver*, de modo que as transferências entre a controladora e a memória podem utilizar DMA. Se a controladora trabalhar dessa maneira, há pouco sentido em ter o *driver* de disco fazendo isso também. Note que ambos, a controladora e o *dri-*

ver, estão em uma boa posição para ler e para gravar trilhas inteiras em um comando, mas que o software independente de dispositivo não pode, porque ele considera um disco como uma seqüência de blocos linear, sem considerar como eles estão divididos em trilhas e em cilindros.

3.7.3 Visão Geral do *Driver* de Disco Rígido no MINIX

O *driver* de disco rígido é a primeira parte do MINIX que vimos que precisa lidar com uma ampla variedade de tipos de hardware diferentes. Antes de discutirmos os detalhes do *driver*, consideraremos brevemente algumas diferenças de hardware que podem causar problemas. O "IBM PC" é realmente uma família de computadores diferentes. Além de processadores diferentes utilizados em membros diferentes da família, também há algumas diferenças importantes no hardware básico. Os membros mais antigos da família, o PC original e o PC-XT, utilizavam um barramento de 8 bits, apropriado para a interface externa de 8 bits do processador 8088. A geração seguinte, o PC-AT, utilizava um barramento de 16 bits, que foi inteligentemente projetado de tal modo que periféricos de 8 bits mais antigos ainda podiam ser utilizados. Contudo, os periféricos de 16 bits mais novos geralmente não podem ser utilizados em sistemas PC-XT mais antigos. O barramento AT foi projetado originalmente para sistemas que utilizavam o processador 80286 e muitos sistemas baseados nos 80386, 80486 e o Pentium utilizam o barramento AT. Entretanto, como esses processadores mais novos têm uma interface de 32 bits, agora há vários sistemas de barramentos de 32 bits diferentes disponíveis, como o barramento PCI da Intel.

Para cada barramento, há uma família diferente de adaptadores de E/S, que são conectados na *parentboard* do sistema. Todo o periférico para um projeto particular de barramento deve ser compatível com os padrões desse projeto mas não precisa ser compatível com projetos mais antigos. Na família IBM PC, como na maioria dos outros sistemas de computador, cada projeto de barramento também vem com *firmware* no *Basic I/O System Read Only Memory* (o BIOS ROM) que é projetado para preencher a lacuna entre o sistema operacional e as peculiaridades do hardware. Alguns dispositivos periféricos podem oferecer extensões para o BIOS em chips de ROM nas próprias placas periféricas. A dificuldade com que se defronta um implementador de sistema operacional é que o BIOS em computadores do tipo IBM (certamente os primeiros) foi projetado para um sistema operacional, MS-DOS, que não suporta multiprogramação e que executa no modo real de 16 bits, o mais baixo denominador comum dos vários modos de operação disponíveis na família de CPUs 80x86.

O implementador de um novo sistema operacional para IBM PC defronta-se, portanto, com várias opções. Uma é utilizar o suporte de *driver* para periféricos no BIOS ou escrever novos *drivers* a partir do zero. Isso não era uma escolha difícil no projeto original do MINIX, uma vez que o BIOS, sob vários aspectos, não era adequado às necessidades do MINIX. Naturalmente, para ser iniciado, o monitor de inicialização do MINIX utiliza o BIOS para fazer a carga inicial do sistema, seja de um disco rígido ou de um disquete — não há uma alternativa prática para fazer isso dessa maneira. Uma vez que carregamos o sistema, incluindo os próprios *drivers* de E/S, podemos fazer muito melhor que o BIOS.

A segunda escolha, então, deve ser encarada: sem o suporte de BIOS como faremos nossos *drivers* adaptarem-se aos vários tipos de hardware em sistemas diferentes? Para tornar a discussão concreta, considere que há pelo menos quatro tipos de controladoras de disco rígido fundamentalmente diferentes que podemos encontrar em um sistema e que, a princípio, seriam adequados ao MINIX: a controladora do tipo XT de 8 bits original, a controladora do tipo AT de 16 bits e duas controladoras diferentes para dois tipos diferentes de computador da série IBM PS/2. Há várias maneiras possíveis de lidar com isso:

1. Recompilar uma versão única do sistema operacional para cada tipo de controladora de disco rígido a que precisamos adaptar-nos.
2. Compilar vários *drivers* de disco rígido diferentes no *kernel* e fazer o *kernel* automaticamente determinar em tempo de inicialização qual utilizar.
3. Compilar vários *drivers* de disco rígido diferentes no *kernel* e proporcionar um meio para o usuário determinar qual utilizar.

Como veremos, tais alternativas não são mutuamente exclusivas.

A primeira é realmente a melhor opção a longo prazo. Para utilização em uma instalação particular não há nenhuma necessidade de consumir espaço em disco e memória com código para *drivers* alternativos que jamais serão utilizados. Entretanto, é um pesadelo para o fabricante do software. Fornecer quatro discos diferentes de inicialização, e instruir os usuários sobre como utilizá-los é caro e difícil. Assim, uma das outras alternativas é aconselhável, pelo menos para a instalação inicial.

O segundo método é fazer o sistema operacional investigar os periféricos, lendo o ROM em cada placa ou gravando e lendo portas de E/S para identificar cada placa. Isso é praticável em alguns sistemas, mas não funciona bem em sistemas tipo IBM porque grande parte dos dispositivos de E/S disponíveis não são padrão. Investigar as portas de E/S para identificar um dispositivo pode, em alguns casos, ativar outro dispositivo que assuma o controle e desative o sistema. Esse método complica o código de inicialização para cada dispositivo e além disso não funciona muito bem. Os sistemas operacionais que utilizam esse método geralmente precisam oferecer algum tipo de sobreposição, normalmente um mecanismo como o que utilizamos no MINIX.

O terceiro método, utilizado no MINIX, é permitir a compilação de vários *drivers*, com um deles sendo o padrão. O monitor de inicialização do MINIX permite que vários **parâmetros de inicialização** sejam lidos em tempo de ini-

cialização. Esses parâmetros podem ser inseridos manualmente ou armazenados permanentemente no disco. Em tempo inicialização, se um parâmetro de inicialização na forma de:

hd = xt

for localizado, isso forçará a utilização do *driver* de disco rígido XT. Se parâmetro de inicialização hd não for encontrado o *driver* padrão é utilizado.

Há duas outras coisas que o MINIX faz para tentar minimizar problemas com múltiplos *drivers* de disco rígido. Uma é que há, no final das contas, um *driver* que faz interface entre o MINIX e o suporte de disco rígido do ROM BIOS. Esse *driver* provavelmente funcionará em qualquer sistema e pode ser selecionado por utilização de um parâmetro

hd = bios

na inicialização. Entretanto, em geral, isso deve ser utilizado como um último recurso. O MINIX executa em modo protegido em sistemas com um processador 80286 ou superior, mas o código de BIOS sempre executa em modo real (8086). Entrar e sair do modo protegido sempre que uma rotina do BIOS é chamada é muito lento.

A outra estratégia que o MINIX utiliza ao lidar com *drivers* é adiar a inicialização até o último momento possível. Assim, se em uma determinada configuração de hardware nenhum dos *drivers* de disco rígido funcionar, ainda podemos iniciar o MINIX a partir de um disquete e fazer algum trabalho útil. O MINIX não terá nenhum problema contanto que não precise acessar o disco rígido. Isso pode não parecer um avanço importante na interface com o usuário, mas considere isso: se todos os *drivers* tentarem iniciar de uma vez ao inicializarmos o sistema, o sistema pode ser totalmente paralisado por configuração imprópria de algum dispositivo do qual não precisaremos no final das contas. Adiando a inicialização de cada *driver* até que seja necessário, o sistema pode continuar com o que funciona, enquanto o usuário tenta resolver os problemas.

A propósito, aprendemos essa lição do jeito mais difícil: versões anteriores do MINIX tentavam iniciar o disco rígido logo que o sistema era inicializado. Se disco rígido não estivesse presente, o sistema era suspenso. Esse comportamento era especialmente infeliz porque o MINIX rodará feliz em um sistema sem disco rígido, embora com capacidade de armazenamento restrita e desempenho reduzido.

Na discussão desta e da próxima seção, tomaremos como modelo o *driver* de disco rígido estilo AT, que é o *driver*-padrão na versão padrão de distribuição do MINIX. Trata-se de um *driver* versátil que gerencia desde as controladoras de disco rígido utilizadas nos antigos sistemas 80286 até as modernas controladoras **EIDE** (***Extended Integrated Drive Electronics***) que gerenciam discos rígidos com capacidades na ordem dos gigabytes. Os aspectos gerais da operação do disco rígido que discutimos nesta seção aplicam-se também aos outros *drivers* suportados.

O laço principal da tarefa de disco rígido é o mesmo código compartilhado que já discutimos, e os seis tipos-padrão de solicitações podem ser feitos. *A* solicitação *DEV_OPEN* pode envolver uma quantidade substancial de trabalho, uma vez que sempre há partições e talvez haja subpartições em um disco rígido. Essas devem ser lidas quando um dispositivo é aberto, (i. e., quando é acessado pela primeira vez). Algumas controladoras de disco rígido também podem suportar unidades de CDROM, que têm mídia removível, e em um *DEV_OPEN* a presença da mídia deve ser verificada. Em um CD-ROM uma operação *DEV_CLOSE* também tem significado: ela requer que a porta seja desbloqueada e o CD-ROM removido. Há outras complicações de mídia removível que são mais aplicáveis a unidades de disquete, portanto, discutiremos essas em uma seção mais adiante. Para o disco rígido a operação *DEV_IOCTL* é utilizada para configurar um sinalizador a fim de assinalar que a mídia que deve ser removida em uma *DEV_CLOSE*. Esse recurso é útil para CD-ROMs. E também é utilizado para ler e para gravar tabelas de partição, como mencionado anteriormente.

As solicitações *DEV_READ, DEV_WRITE* e *SCATTERED_IO* são tratadas em três fases, preparação, agendamento e término, como vimos. O disco rígido, ao contrário dos dispositivos de memória, faz uma distinção real entre as fases de agendamento e término. O *driver* de disco rígido não utiliza SSF nem o algoritmo do elevador, mas realiza uma forma mais limitada de agendamento, reunindo solicitações a setores consecutivos. As solicitações normalmente vêm do sistema de arquivos MINIX e são para múltiplos blocos de 1024 bytes, mas o *driver* é capaz de tratar solicitações para qualquer múltiplo de um setor (512 bytes). Contanto que cada solicitação seja para um setor imediatamente seguinte ao último setor requisitado, cada solicitação é anexada a uma lista de solicitações. A lista é mantida como uma matriz e quando estiver cheia ou quando um setor não-consecutivo for solicitado, é feita uma chamada para a rotina de término.

Em uma simples solicitação *DEV_READ* ou *DEV_WRITE*, mais de um bloco pode ser solicitado, mas cada chamada à rotina de agendamento é imediatamente seguida por uma chamada à rotina de término, o que assegura que a lista atual de solicitações seja atendida. No caso de uma solicitação *SCATTERED_IO*, pode haver múltiplas chamadas à rotina de agendamento antes de a rotina de término ser chamada. Contanto que seja para blocos consecutivos de dados, a lista será estendida até que a matriz esteja cheia. Lembre-se de que, em uma solicitação *SCATTERED_IO*, um sinalizador pode significar que uma solicitação para um bloco particular é opcional. O *driver* de disco rígido, como o *driver* de memória, ignora o sinalizador *OPTIONAL* e entrega todos os dados *solicitados*.

O agendamento rudimentar realizado pelo *driver* de disco rígido, adiando as transferências reais enquanto blocos consecutivos estão sendo solicitados, deve ser visto como o segundo passo de um potencial processo de três passos de

agendamento. O próprio sistema de arquivos, utilizando E/S dispersa, pode implementar algo semelhante para a versão de Teory do algoritmo do elevador — lembre-se de que em uma E/S dispersa a lista de solicitações é classificada segundo o número de bloco. O terceiro passo no agendamento acontece na controladora de um disco rígido moderno, como o descrito na Figura 3-19. Essas controladoras são "espertas" e podem *bufferizar* grandes quantidades de dados, utilizando algoritmos programados internamente para recuperar dados na ordem mais eficiente, independentemente da ordem de recebimento das solicitações.

3.7.4 A Implementação do *Driver* de Disco Rígido no MINIX

Discos rígidos pequenos utilizados em microcomputadores, às vezes, são chamados *winchesters*. Há várias histórias diferentes sobre a origem do nome. Aparentemente, era um nome de código da IBM para o projeto que desenvolveu a tecnologia de disco em que os cabeçotes de leitura/gravação flutuam sobre uma fina almofada de ar e pousam na mídia de gravação quando o disco pára de girar. Uma explicação do nome é que um modelo anterior tinha dois módulos de dados, um de 30 Mbytes fixo e um removível de 30 Mbytes. Supostamente isso lembrava aos desenvolvedores as antigas armas Winchester 30-30 que figuravam em muitas histórias de faroeste. Qualquer que seja a origem do nome, a tecnologia básica permanece a mesma, embora hoje um típico disco de microcomputador seja muito menor e a capacidade seja muito maior do que os discos de 14 polegadas que eram típicos no começo da década de 70 quando a tecnologia dos *winchesters* foi desenvolvida.

O arquivo *wini.c* tem o trabalho de ocultar o *driver* de disco rígido utilizado do resto do *kernel*. Isso nos permite seguir a estratégia discutida na seção anterior, compilando vários *drivers* de disco rígido em uma única imagem de *kernel* e selecionar um em tempo de inicialização. Mais tarde, uma instalação personalizada pode ser recompilada com apenas o *driver* realmente necessário.

Wini.c contém uma definição de dados, *hdmap* (linha 10013), uma matriz que associa um nome com o endereço de uma função. A matriz é iniciada pelo compilador com quantos elementos forem necessitados pelo número de *drivers* de disco rígido ativados em *included/minix/config.h*. A matriz é utilizada pela função *winchester_task*, que é o nome inserido na tabela *task_tab* utilizada quando o *kernel* é iniciado pela primeira vez. Quando *winchester_task* (linha 10040) é chamada, ela tenta localizar uma variável de ambiente *bd*, utilizando uma função de *kernel* que funciona de maneira semelhante ao mecanismo utilizado por programas normais em C, lendo o ambiente criado pelo monitor de inicialização do MINIX. Se o valor de *bd* não estiver definido, a primeira entrada na matriz será utilizada; caso contrário, ela é pesquisada quanto a um nome que coincida. A função correspondente, então, é chamada de maneira indireta. No restante desta seção, discutiremos o *at_winchester_task*, que é a primeira entrada na matriz *hdmap* na distribuição padrão do MINIX.

O *driver* estilo AT está em *at_wini.c* (linha 10100). Esse é um *driver* complicado para um dispositivo sofisticado e há várias páginas de definições de macros para especificar registradores de controladora, bits de status e comandos, estruturas de dados e protótipos. Como com outros *drivers* de dispositivo de bloco, uma estrutura *driver*, *w_dtab* (linhas 10274 para 10284) é iniciada com ponteiros para as funções que realmente fazem o trabalho. A maioria delas é definida em *at_wini.c*, mas como o disco rígido não requer nenhuma operação especial de limpeza, sua entrada *dr_cleanup* aponta para *nop_cleanup* comum em *driver.c*, compartilhado com outros *drivers* que não fazem nenhuma exigência especial de limpeza. A função de entrada, *at_winchester_task* (linha 10294), chama um procedimento que faz inicialização específica de hardware e, então, chama o laço principal em *driver.c*. Este último executa eternamente, despachando chamadas para as várias funções apontadas pela tabela *driver*.

Como agora estamos lidando com dispositivos de armazenamento eletromecânicos reais, há uma quantidade substancial de trabalho a ser feito para iniciar o *driver* de disco rígido. Vários parâmetros sobre os discos rígidos são mantidos na matriz *wini* definida nas linhas 10214 a 10230. Como parte da política de adiar passos de inicialização que possam falhar até quando eles forem realmente necessários, *init_params* (linha 10307), que é chamado durante a inicialização do *kernel*, não faz nada que exija acessar o dispositivo de disco. A principal coisa que ele faz é copiar algumas informações sobre a configuração lógica do disco rígido na matriz *wini*. Essas são as informações recuperadas pelo ROM BIOS da memória CMOS que os computadores da classe Pentium utilizam para preservar dados básicos de configuração. As ações do BIOS acontecem quando o computador é ligado, antes de a primeira parte do processo de carregamento do MINIX começar. Não é necessariamente fatal se tais informações não puderem ser recuperadas; se o disco for um disco moderno, as informações podem ser recuperadas diretamente dele.

Após a chamada ao laço principal comum, nada pode acontecer temporariamente até que uma tentativa seja feita para acessar o disco rígido. Então, uma mensagem solicitando uma operação *DEV_OPEN* é recebida e *w_do_open* (linha 10355) é indiretamente chamada. Por sua vez, *w_do_ open* chama *w_prepare* para determinar se o dispositivo solicitado é válido, e então *w_identify* para identificar o tipo de dispositivo e iniciar alguns parâmetros adicionais na matriz *wini*. Por fim um contador na matriz *wini* é utilizado para testar se essa é a primeira vez que o dispositivo foi aberto desde que o MINIX foi iniciado. Depois de ser examinado, o contador é incrementado. Se essa for a primeira operação *DEV_OPEN*, a função *partition* (em *drvlib.c*) é chamada.

A próxima função, *w_prepare* (linha 10388), aceita um argumento de número inteiro, *device*, que é o número de dispositivo secundário da unidade ou da partição a ser uti-

lizado e retorna um ponteiro para a estrutura *device* que indica o endereço de base e o tamanho do dispositivo. Em C, a utilização de um identificador para nomear uma estrutura não impede a utilização do mesmo identificador para nomear uma variável. Se um dispositivo é uma unidade, uma partição ou subpartição pode ser determinado a partir do número do dispositivo secundário. Uma vez que *w_prepare* tenha completado seu trabalho, nenhuma das outras funções utilizadas para ler ou para gravar no disco precisa preocupar-se com o particionamento. Como vimos, *w_prepare* é chamada quando uma solicitação *DEV_OPEN* é feita; também é uma fase do ciclo preparação/agendamento/término utilizado por toda solicitação de transferência de dados. Nesse contexto sua inicialização de *w_count* para zero é importante.

Discos compatíveis com software do tipo AT estiveram em utilização por um bom tempo e *w_identify* (linha 10415) precisa distinguir entre diversos projetos diferentes que foram apresentados ao longo dos anos. O primeiro passo é ver se existe uma porta de E/S legível e gravável onde uma deveria existir em todas as controladoras de disco nessa família (linhas 10435 a 10437). Se essa condição for satisfeita, o endereço do manipulador de interrupções do disco rígido é instalado na tabela de descritores de interrupções, e o controlador de interrupções é ativado para responder a essa interrupção. Então, um comando *ATA_IDENTIFY* é enviado para a controladora de disco. Se o resultado for *OK*, várias informações são recuperadas, incluindo uma *string* que identifica o modelo do disco e os parâmetros físicos de cilindro, cabeçote e setor para o dispositivo. (Note que a configuração "física" informada pode não ser a configuração física verdadeira, mas não temos nenhuma alternativa a não ser aceitar o que a unidade de disco declara.) As informações de disco também indicam se o disco é capaz de **Endereçamento Linear de Bloco (LBA — *Linear Block Addressing*)**. Se for, o *driver* poderá ignorar os parâmetros de cilindro, cabeçote e setor e pode endereçar o disco utilizando números absolutos de setor, que é muito mais simples.

Como mencionamos anteriormente, é possível que *init_params* não possa recuperar a configuração lógica de disco das tabelas de BIOS. Se isso acontecer, o código nas linhas 10469 a 10477 tentará criar um conjunto apropriado de parâmetros com base no que ele lê da própria unidade. A idéia é que os números máximos de cilindro, cabeçote e setor podem ser 1023, 255 e 63 respectivamente, devido ao número de bits permitido para esses campos nas estruturas de dados originais do BIOS.

Se o comando *ATA_IDENTIFY* falhar, isso pode significar simplesmente que o disco é um modelo mais antigo que não suporta o comando. Nesse caso, os valores lógicos de configuração previamente lidos por *init_params* é tudo que temos. Se forem válidos, eles são copiados para os campos de parâmetro físicos de *wini;* Caso contrário, um erro é retornado, e o disco não é utilizável.

Por fim, o MINIX utiliza uma variável *u32_t* para contar os endereços em bytes. O tamanho de dispositivo que o *driver* pode tratar, expresso como uma contagem de setores, deve ser limitado se o produto de cilindros × cabeçotes × setores for muito grande (linha 10490). Embora à época em que este código foi escrito dispositivos de capacidade de 4 GB raramente fossem encontrados em máquinas onde se poderia esperar que o MINIX fosse utilizado, a experiência ensinou que se deve escrever software para testar limites como esse, por mais que esses testes pudessem parecer desnecessários à época em que o código foi escrito. A base e o tamanho da unidade inteira, então, são inseridos na matriz *wind,* e *w_specify* é chamada, duas vezes se necessário, para passar de volta à controladora de disco os parâmetros a serem utilizados. Por fim, o nome do dispositivo (determinado por *w_name*) e a *string* de identificação localizada por *identify* (se for um dispositivo avançado) ou os parâmetros de cabeçote, cilindro e setor informados pelo BIOS (se for um dispositivo antigo) são impressos no console.

W_name (linha 10511) retorna um ponteiro para uma *string* que contém o nome do dispositivo, que será "at-hd0", "at-hd5", "at-hd10" ou "at-hd15". *W_specify* (linha 10531), além de passar os parâmetros à controladora, também recalibra a unidade (se é um modelo mais antigo), fazendo uma busca para o cilindro zero.

Agora estamos prontos para discutir as funções chamadas para atender uma solicitação de transferência de dados. *W_prepare,* que já discutimos, é chamada primeiro. Sua inicialização da variável *w_count* para zero é importante aqui. A próxima função chamada durante uma transferência é *w_schedule* (linha 10567). Ela configura os parâmetros básicos: de onde os dados vêm, para onde irão, a contagem de bytes a transferir (que deve ser um múltiplo do tamanho de setor, o que é testado na linha 10584) e se a transferência é uma leitura ou uma gravação. O bit que pode estar presente em uma solicitação *SCATTERED_IO* para indicar uma transferência opcional é redefinido no código de operação a ser passado para a controladora (linha 10595), mas note que ele é retido no campo *io_request* da estrutura *iorequest_s*. Para o disco rígido, é feita uma tentativa de atender todas as solicitações, mas, como veremos, o *driver* pode, mais tarde, decidir não fazer isso se houver erros. A última coisa na configuração é verificar se a solicitação não vai além do último byte no dispositivo e reduzir a solicitação se for. Nesse ponto, o primeiro setor a ser lido pode ser calculado.

Na linha 10602, o processo de agendamento começa realmente. Se já houver solicitações pendentes (testado vendo-se se *w_count* é maior que zero) e se o setor a ser lido em seguida não é consecutivo ao último solicitado, então *w_finish* é chamada para completar as solicitações anteriores. Caso contrário, *w_nextblock,* que armazena o número de setor do próximo setor, é atualizada, e o laço nas linhas 10611 a 10640 é iniciado para adicionar novas solicitações de setor à matriz de solicitações. O laço continua

até que o número máximo admissível de solicitações seja atingido (linha 10614). O limite é mantido em uma variável, *max_count*, uma vez que, como veremos mais adiante, às vezes, é útil ser possível de ajustar o limite. Aqui novamente pode ocorrer uma chamada para *w_finish*.

Como vimos, há dois lugares dentro de *w_prepare* nos quais uma chamada a *w_finish* pode ser feita. Normalmente *w_prepare* termina sem chamar *w_finish*, mas seja ou não chamada a partir de *w_prepare*, *w_finish* (linha 10649) sempre acaba sendo chamada a partir do laço principal em *driver.c*. Se ela acabou de ser chamada, pode não ter nada a fazer; então, há um teste na linha 10659 para verificar isso. Se ainda houver solicitações na matriz de solicitações, a parte principal de *w_finish* é iniciada.

Como se poderia esperar, uma vez que pode haver um número considerável de solicitações enfileiradas, a parte principal de *w_finish* é um laço, nas linhas 10664 a 10761. Antes de iniciar o laço, a variável *r* é predefinida para um valor que significa um erro, para forçar a reinicialização da controladora. Se uma chamada a *w_specify* sucede à estrutura *command*, *cmd* é inicializado para fazer uma transferência. Essa estrutura é utilizada para passar todos os parâmetros necessários à função que realmente opera a controladora de disco. O parâmetro *cmd.precomp* é utilizado por algumas unidades para compensar algumas diferenças no desempenho de gravação da mídia magnética com diferenças na velocidade da passagem da mídia sob os cabeçotes de disco à medida que eles se movem dos cilindros mais exteriores para os mais interiores. Ele é sempre o mesmo para uma unidade particular e é ignorado por muitas unidades. *Cmd.count* recebe o número de setores a transferir, mascarado para uma quantidade que se ajusta em um byte de 8 bits, uma vez que este é o tamanho de todos os registradores de comandos e de status da controladora. O código nas linhas 10675 a 10689 especifica o primeiro setor a transferir, seja como um número de bloco lógico de 28 bits (linhas 10676 a 10679) ou como parâmetros de cilindro, cabeçote e setor (linhas 10681 a 10688). Em qualquer caso, os mesmos campos na estrutura *cmd* são utilizados.

Por fim, o próprio comando, de leitura ou gravação, é carregado e *com_out* é chamada na linha 10692 para iniciar a transferência. A chamada *com_out* pode falhar se a controladora não estiver pronta ou não ficar pronta dentro de um período predefinido. Nesse caso, a contagem de erros é incrementada, e a tentativa é abortada se *MAX_ERRORS* for alcançado. Caso contrário, a declaração

```
continue;
```

na linha 10697 faz o laço iniciar novamente na linha 10665.

Se a controladora aceitar o comando passado na chamada para *com_out*, pode demorar um pouco até os dados estarem disponíveis, portanto (supondo que o comando é *DEV_READ*) *w_intr_wait* é chamada na linha 10706. Discutiremos essa função detalhadamente mais tarde, mas, por enquanto, observe apenas que ela chama *receive*, portanto, neste ponto a tarefa de disco é bloqueada.

Algum tempo mais tarde, mais ou menos, dependendo de uma busca estar ou não envolvida, a chamada a *w_intr_wait* retornará. Esse *driver* não utiliza DMA, embora algumas controladoras suportem-no. Em vez disso, é utilizada E/S programada. Se não houver nenhum erro retornado de *w_intr_wait*, a função de linguagem de *assembly port_read* irá transferir *SECTOR_SIZE* bytes de dados da porta de dados da controladora para seu destino final, que deve ser um buffer no *cache* de blocos do sistema de arquivos. Em seguida, vários endereços e contagens são ajustados para registrar a transferência bem-sucedida. Por fim, se a contagem de bytes na solicitação atual for para zero, o ponteiro para a matriz de solicitações é avançado para apontar para a próxima solicitação (linha 10714).

No caso de um comando *DEV_WRITE*, a primeira parte, configurar os parâmetros de comando e enviar o comando à controladora, é a mesma que para uma leitura, exceto para o código de operação do comando. Entretanto, a ordem dos eventos subseqüentes é diferente para uma gravação. Primeiro, há uma espera para a controladora sinalizar que está pronta para receber os dados (linha 10724). *Waitfor* é uma macro e normalmente retornará muito rapidamente. Falaremos mais sobre ela mais tarde; por enquanto, apenas observaremos que a espera poderá eventualmente atingir o tempo-limite, mas retardos longos são extremamente raros. Então, os dados são transferidos da memória para a porta de dados da controladora utilizando *port_write* (linha 10729) e neste ponto *w_intr_wait* é chamada, e a tarefa de disco é bloqueada. Quando a interrupção chega e a tarefa de disco é acordada, a contabilização é feita (linhas 10736 a 10739).

Por fim, se houve erros na leitura ou gravação, eles devem ser tratados. Se a controladora informa o *driver* de que o erro foi devido a um setor defeituoso, não há nenhum sentido em tentar novamente, mas outros tipos de erro merecem uma nova tentativa, pelo menos até um ponto. Esse ponto é determinado contando-se os erros e desistindo se *MAX_ERRORS* for alcançado. Quando *MAX_ERRORS/2* é alcançado, *w_need_reset* é chamada para forçar a reinicialização quando a nova tentativa for feita. Entretanto, se a solicitação era originalmente opcional (feita por uma solicitação *SCATTERED_IO*), nenhuma nova tentativa é feita.

Se *w_finish* terminar sem erros ou devido a um erro, a variável *w_command* é sempre configurada como *CMD_IDLE*. Isso permite que outras funções determinem se a falha não se deveu a um mau funcionamento mecânico ou elétrico do próprio disco causando uma falha na geração de uma interrupção após uma tentativa de operação.

A controladora de disco é controlada por um conjunto de registradores, que podem ser mapeados em memória em alguns sistemas, mas em um sistema compatível com IBM aparecem como portas de E/S. Os registradores utilizados por uma controladora de disco rígido padrão da classe de IBM-AT são mostrados na Figura 3-22.

Este é nosso primeiro encontro com hardware de E/S e pode ser útil mencionar que algumas portas de E/S com-

portam-se diferentemente dos endereços de memória. Em geral, registradores de entrada e saída que eventualmente têm o mesmo endereço de porta de E/S não são o mesmo registrador. Assim, os dados gravados em um endereço particular podem não ser necessariamente recuperados por uma subseqüente operação de leitura. Por exemplo, o último endereço de registrador mostrado na Figura 3-22 mostra o status da controladora de disco durante uma leitura e é utilizado para enviar comandos para a controladora durante uma gravação. Também se sabe que o simples ato de ler e de escrever em um registrador de um dispositivo de E/S causa a ocorrência de uma ação, independentemente dos detalhes dos dados transferidos. Isso é verdade quanto ao registrador de comando na controladora de disco AT. Em uso, dados são gravados nos registradores de numeração inferior para selecionar o endereço do disco a ser lido ou gravado e, então, gravar-se no registrador de comando um código de operação. O ato de gravar o código de operação no registrador de comando inicia a operação.

Esse também é o caso em que a utilização de alguns registradores ou de campos nos registradores pode variar de acordo com os diferentes modos de operação. No exemplo dado na figura, a gravação de um 0 ou um 1 no bit de LBA, bit 6 do registrador 6, seleciona se é utilizado o modo CHS (cilindro, cabeçote e setor – *Cylinder-Head-Sector*) ou LBA. Os dados gravados ou lidos dos registradores 3, 4 e 5 e os quatro bits inferiores do registrador 6 são interpretados diferentemente de acordo com a configuração do bit de LBA.

Agora vejamos como um comando é enviado para a controladora chamando *com_out* (linha 10771). Antes de mudar qualquer registrador, o registrador de status é lido para determinar se a controladora não está ocupada. Isso é feito testando o bit *STATUS_BSY*. A velocidade é importante aqui e normalmente a controladora de disco está pronta ou estará pronta em breve, assim é utilizada espera ativa. Na linha 10779, *waitfor* é chamada para testar *STATUS_BSY*. Para maximizar a velocidade de resposta, *waitfor* é uma macro, definida na linha 10268. Ela faz o teste necessário

Registrador	Função *Read*	Função *Write*
0	Dados	Dados
1	Erro	Write Precompensation
2	Contagem de Setores	Contagem de Setores
3	Número do Setor (0-7)	Número do Setor (0-7)
4	Cilindro (Baixo) (8-15)	Cilindro (Baixo) (8-15)
5	Cilindro (Alto) (16-23)	Cilindro (Alto) (16-23)
6	Seleção de Unidade/Cabeçote (24-27)	Seleção de Unidade/Cabeçote (24-27)
7	Status	Comando

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Modo de cilindro/cabeçote/setor (CHS)
1 = Modo de endereçamento de bloco lógico (LBA)

D: 0 = unidade mestre
1 = unidade escrava

HSn: Modo CHS: Seleciona cabeçote, no modo CHS
Modo LBA: Seleciona bloco (bits 24 - 27)

(b)

Figura 3-22 (a) Os registradores de controle de uma controladora de disco rígido IDE. Os números entre parênteses são os bits do endereço de bloco lógico selecionados por cada registrador no modo LBA. (b) Os campos do registrador *Seleção de Unidade/Cabeçote*.

uma vez, evitando uma cara chamada de função na maioria das chamadas, quando o disco está pronto. Nas raras ocasiões em que uma espera é necessária, ela então chama *waitfor*, que executa o teste em um laço até que seja verdadeiro ou até que se passe um limite de tempo predefinido. Assim, o valor retornado será verdadeiro com o mínimo atraso possível se a controladora estiver pronta, verdadeiro depois de um retardo se ela estiver temporariamente indisponível ou falso se ela não estiver pronta depois de esgotado o limite de tempo. Teremos mais a dizer sobre o limite de tempo ao discutirmos a própria *w_waitfor*.

Uma controladora pode tratar mais de uma unidade; então, uma vez que se determine que a controladora está pronta, um byte é gravado para selecionar a unidade, o cabeçote e o modo de operação (linha 10785) e, então, *waitfor* é chamada novamente. Às vezes, uma unidade de disco não consegue executar um comando ou retornar um código de erro adequadamente — ela é, afinal de contas, um dispositivo mecânico que pode emperrar, enroscar ou quebrar internamente — e como medida de segurança uma mensagem é enviada para a tarefa de relógio agendar uma chamada para uma rotina de *wakeup*. Depois disso, é enviado o comando gravando-se primeiro todos os parâmetros nos vários registradores e por fim gravando o próprio código do comando no registrador de comando. Este último passo e a modificação subseqüente das variáveis *w_command* e *w_status* é uma seção crítica; então, a seqüência inteira é envolvida por chamadas a *lock* e *unlock* (linhas 10801 a 10805) que desativam e, então, reativam as interrupções.

As próximas funções são curtas. Notamos que *w_need_reset* (linha 10813) é chamada por *w_finish* quando a contagem de erros atinge metade de *MAX_ERRORS*. Ela também é chamada quando são atingidos limites de tempo enquanto se espera o disco gerar uma interrupção ou tornar-se pronto. A ação de *w_need_reset* é simplesmente marcar a variável *state* para cada unidade na matriz *wini* para forçar a inicialização no próximo acesso.

W_do_close (linha 10828) tem muito pouco a fazer para um disco rígido convencional. Quando se acrescentar suporte a CD-ROMs ou outros dispositivos removíveis, essa rotina terá de ser estendida a fim de gerar um comando para destravar a porta ou para ejetar um CD, dependendo do que o hardware suporte.

Com_simple é chamada para enviar comandos de controladora que terminam imediatamente sem uma fase de transferência de dados. Comandos que entram nessa categoria incluem os que recuperam a identificação de disco, configuram alguns parâmetros e fazem recalibragem.

Quando *com_out* chama a tarefa de relógio para preparar um possível salvamento depois de uma falha de uma controladora de disco, ele passa o endereço de *w_timeout* (linha 10858) como a função para a tarefa de relógio acordar quando o tempo-limite expirar. Normalmente o disco completa a operação requerida e quando o tempo-limite ocorre, *w_command* será encontrado como o valor *CMD_IDLE*, o que significa que o disco completou sua operação e *w_timeout*, então, pode terminar. Se o comando não se completar e a operação for uma leitura ou uma gravação, reduzir o tamanho das solicitações de E/S pode ajudar. Isso é feito em duas etapas, primeiro reduzindo o número máximo de setores que podem ser solicitados para 8 e depois para 1. Para todos os limites de tempo uma mensagem é impressa; *w_need_reset* é chamada para forçar a reinicialização de todas as unidades na próxima tentativa de acesso e *interrupt* é chamada para entregar uma mensagem para a tarefa de disco e para simular interrupções geradas por hardware que devem ocorrer no final da operação de disco.

Quando uma reinicialização é exigida, *w_reset* (linha 10889) é chamada. Essa função faz uso de uma função fornecida pelo *driver* de relógio, *milli_delay*. Depois de um retardo inicial para dar tempo para a unidade recuperar-se de operações anteriores, um bit no registrador de controle da controladora de disco é sinalizado — isto é, ele é trazido para um nível lógico 1 durante um período definido, e então, retornado para o nível lógico 0. Seguindo-se a essa operação, *waitfor* é chamada para dar um tempo razoável para a unidade sinalizar que está pronta. Caso a reinicialização não tenha sucesso, uma mensagem é impressa, e um código de erro é retornado. Deixa-se o processo que fez a chamada decidir o que fazer em seguida.

Comandos para disco que envolvem transferência de dados normalmente terminam gerando uma interrupção, que envia uma mensagem de volta à tarefa de disco. De fato, uma interrupção é gerada para cada setor lido ou gravado. Assim, depois de enviar esse tipo de comando, *w_intr_wait* (linha 10925) sempre será chamada. Por sua vez, *w_intr_wait* chama *receive* em um laço, ignorando o conteúdo de cada mensagem, esperando uma interrupção que configure *w_status* como não-ocupada. Uma vez recebida essa mensagem, o status da solicitação é verificado. Essa é outra seção crítica; então, *lock* e *unlock* são utilizados aqui para garantir que uma nova interrupção não ocorrerá e mudará *w_status* antes de os vários passos envolvidos estarem completos.

Vimos vários lugares onde a macro *waitfor* é chamada para fazer espera ativa em um bit no registrador de status da controladora de disco. Depois do teste inicial, a macro *waitfor* chama *w_waitfor* (linha 10955), que chama *milli_start* para iniciar um temporizador e então entra em um laço que alternadamente verifica o registrador de status e o temporizador. Se um limite de tempo é atingido, *w_need_reset* é chamada para configurar as coisas para uma reinicialização da controladora de disco da próxima vez que seus serviços forem solicitados.

O parâmetro *TIMEOUT* utilizado por *w_waitfor* é definido na linha 10206 como 32 segundos. Um parâmetro semelhante, *WAKEUP* (linha 10193), utilizado para agendar *wakeups* da tarefa de relógio, é configurado como 31 segundos. Esses são períodos de tempo muito longos para uma espera ariva, quando se considera que um processo comum recebe apenas 100ms para executar antes de ser removido. Mas esses números são baseados no padrão pu-

blicado para interfaces de dispositivos de disco com computadores da classe AT, a qual sustenta que até 31 segundos devem ser permitidos para um disco entrar em rotação. O fato é que isso, naturalmente, é uma especificação do pior cenário e que na maioria dos sistemas o disco entra em rotação logo na partida, ou talvez depois de longos períodos de inatividade. O MINIX ainda está sendo desenvolvido. Portanto, é possível que uma nova maneira de tratar limites de tempo seja indicada quando se acrescentar suporte para CD-ROMs (ou outros dispositivos que necessitam iniciar a rotação com freqüência).

W_handler (linha 10976) é o manipulador de interrupções. O endereço dessa função é colocado na Tabela de Descritores de Interrupções por *w_identify* quando a tarefa de disco rígido é ativada pela primeira vez. Quando ocorre uma interrupção de disco, o registrador de status da controladora de disco é copiado para *w_status* e então a função *interrupt* no *kernel* é chamada para reagendar a tarefa de disco rígido. Quando isso ocorre, naturalmente, a tarefa de disco rígido já está bloqueada como resultado de uma chamada a *receive* ou *w_intr_wait* depois do início de uma operação de disco.

A última função em *at_wini.c* é *w_geometry*. Ela retorna os valores lógicos máximos de cilindro, cabeçote e setor do dispositivo de disco rígido selecionado. Nesse caso, os números são reais, não mascarados como foram para o *driver* de disco de RAM.

3.7.5 Manipulação de Disquete

O *driver* de disquete é mais longo e mais complicado do que o *driver* de disco rígido. Isso pode parecer paradoxal, uma vez que mecanismos de disquete poderiam parecer mais simples do que os de discos rígidos, mas o mecanismo mais simples tem uma controladora mais simples que requer mais atenção do sistema operacional; e o fato de a mídia ser removível adiciona algumas complicações. Nesta seção descreveremos algumas coisas que um implementador precisa considerar ao lidar com disquetes. Entretanto, não entraremos nos detalhes do código do *driver* de disquete do MINIX. As partes mais importantes são semelhantes àquelas para o disco rígido.

Uma das coisas com que não temos de preocupar-nos em relação ao *driver* de disquete é o suporte a múltiplos tipos de controladora com que temos de lidar no caso do *driver* de disco rígido. Embora os disquetes de alta densidade atualmente utilizados não fossem suportados no projeto do IBM PC original, as controladoras de disquete de todos os computadores da família IBM PC são suportadas por um único *driver* de software. O contraste com a situação dos discos rígidos provavelmente se deve à falta de pressão para aumentar o desempenho dos disquetes. Os disquetes raramente são utilizados como mídia de armazenamento de trabalho durante a operação de um sistema de computador; sua velocidade e capacidade de dados também são limitadas se comparadas com as de discos rígidos. Os disquetes permanecem importantes para distribuição de novo software e para backup, assim quase todos os sistemas de computador de pequeno porte são equipados com pelo menos uma unidade de disquetes.

O *driver* de disquete não utiliza SSF ou algoritmo do elevador. Ele é estritamente seqüencial, aceitando uma solicitação e executando-a antes de aceitar outra solicitação. No projeto original do MINIX foi sentido que, como o MINIX foi destinado para utilização em computadores pessoais, a maioria das vezes haveria somente um processo ativo, e a chance de uma solicitação de disco chegar enquanto outra estivesse sendo executada era pequena. Assim, haveria pouco a ganhar com o aumento considerável na complexidade do software que seria necessário para enfileirar solicitações. É até menos vantajoso agora, uma vez que os disquetes raramente são utilizados para qualquer coisa além de transferência de dados para dentro e para fora de um sistema com um disco rígido.

Dito isso, mesmo que não haja nenhum suporte no software de *driver* para reorganizar as solicitações, o *driver* de disquete, como qualquer outro *driver* de bloco, pode tratar uma solicitação de E/S dispersa e, assim, como o *driver* de disco rígido, o *driver* de disquete coleciona solicitações em uma matriz e continua a colecionar essas solicitações, contanto que setores seqüenciais sejam solicitados. Entretanto, no caso do *driver* de disquete, a matriz de solicitações é menor que para o disco rígido, limitada ao número máximo de setores por trilha em um disquete. Além disso, o *driver* de disquete presta atenção ao sinalizador *OPTIONAL* em uma solicitação de E/S dispersa e não prossegue para uma nova trilha se as solicitações atuais forem opcionais.

A simplicidade do hardware de disquete é responsável por algumas complicações no software do *driver* de disquete. Unidades de disquete de baixa capacidade, lentas e baratas não justificam as controladoras integradas sofisticadas que são parte dos discos rígidos modernos; então, o software do *driver* precisa negociar explicitamente com aspectos de operação de disco que estão ocultos na operação de um disco rígido. Como um exemplo de uma complicação causada pela simplicidade das unidades de disquete, considere o posicionamento do cabeçote de leitura/gravação sobre uma trilha particular durante uma operação SEEK. Nenhum disco rígido jamais exigiu que o software de *driver* explicitamente chamasse uma SEEK. Para um disco rígido, o cilindro, o cabeçote e a geometria de setor visível para o programador podem não corresponder à geometria física, e, de fato, a geometria física pode ser bem complicada, com mais setores nos cilindros exteriores do que nos interiores. Isso, porém, não é visível para o usuário. Discos rígidos podem aceitar *Logical Block Addressing* (LBA), endereçando pelo número absoluto de setor no disco, como uma alternativa para o endereçamento por cilindro, cabeçote e setor. Mesmo que o endereçamento seja feito por cilindro, cabeçote e setor, qualquer geometria que não enderece setores inexistentes pode ser utilizada, desde

que a controladora integrada no disco calcule para onde mover os cabeçotes de leitura/gravação e faça uma operação de busca quando necessário.

Para um disquete, entretanto, é necessária programação explícita de operações SEEK. No caso de uma SEEK falhar, é necessário oferecer uma rotina para executar a operação RECALIBRATE, que força os cabeçotes para o cilindro 0. Isso torna possível para a controladora avançá-los para uma posição desejada de trilha, parando os cabeçotes por um número conhecido de vezes. Operações semelhantes são necessárias para o disco rígido, naturalmente, mas a controladora da unidade trata-os sem orientação detalhada do software do *driver* de dispositivo.

Algumas características de uma unidade de disquete que complicam seu *driver* são:

1. Mídia Removível.
2. Múltiplos Formatos de Disco.
3. Controle de Motor.

Algumas controladoras de disco rígido oferecem mídia removível, por exemplo, em uma unidade de CD-ROM, mas a controladora da unidade geralmente é capaz de tratar quaisquer complicações sem muito suporte no software de *driver* de dispositivo. Com um disquete, entretanto, não há o suporte interno e ainda é necessário mais. Algumas utilizações mais comuns para disquetes — instalar novo software ou fazer backup de arquivos — podem exigir troca dos discos nas unidades. Isso pode causar problemas se os dados que foram destinados a um disquete forem gravados em outro disquete. O *driver* de dispositivo deve fazer o que puder para evitar isso, embora isso nem sempre seja possível, uma vez que nem todo hardware de unidade de disquete permite determinar se a porta de unidade foi aberta desde o último acesso. Outro problema que pode ser causado por mídia removível é que um sistema poderá ser suspenso se for feita uma tentativa de acessar uma unidade de disquete que atualmente não tem nenhum disquete inserido. Isso pode ser resolvido se uma porta aberta puder ser detectada, mas como isso nem sempre é possível, alguma provisão deve ser feita para um tempo limite, e um retorno de erro se uma operação em um disquete não terminar em um tempo razoável.

Mídia removível pode ser substituída por outra mídia, e no caso dos disquetes há muitos possíveis formatos diferentes. O hardware do MINIX suporta unidades de disco tanto de 3,5 pol. como de 5,25 pol. e os disquetes podem ser formatados em uma variedade de maneiras para armazenar de 360KB a 1,2MB (em um disquete 5,25 pol.) ou 1,44MB (em um disquete de 3,5 pol.). O MINIX suporta sete formatos diferentes de disquetes. Há duas possíveis soluções para o problema que isso causa e o MINIX permite ambas. Uma maneira é referir a cada possível formato como uma unidade distinta e oferecer múltiplos dispositivos secundários. O MINIX faz isso e no diretório de dispositivos você encontrará 14 dispositivos diferentes definidos, variando de */dev/pc0,* um disquete de 5,25 pol. 360K na primeira unidade, a */dev/PS1*, um disquete de 3,5 pol. 1.44M na segunda unidade. Lembrar das combinações diferentes é incômodo, e uma segunda alternativa é fornecida. Quando a primeira unidade de disquete é endereçada como */dev/fd0*, ou a segunda como */dev/fd1*, o *driver* de disquete testa o disquete atualmente na unidade quando é acessado, para determinar o formato. Alguns formatos têm mais cilindros e outros têm mais setores por trilha. A determinação do formato de um disquete é feita tentando-se ler os setores e as trilhas numerados mais altos. Por um processo de eliminação o formato pode ser determinado. Isso, naturalmente, leva tempo, e um disco com setores defeituosos poderia ser mal-identificado.

A complicação final do *driver* de disquete é o controle do motor. Os disquetes não podem ser lidos nem gravados a menos que estejam girando. Discos rígidos são projetados para rodar milhares de horas sem se desgastarem, mas deixar os motores ligados o tempo todo faz com que uma unidade de disquetes desgaste-se rapidamente. Se o motor ainda não estiver ligado quando uma unidade é acessada, é necessário dar um comando para acionar a unidade e, então, esperar cerca de meio segundo antes de tentar ler ou gravar dados. Ligar e desligar os motores é lento; então, o MINIX deixa o motor de uma unidade ligado por alguns segundos depois que uma unidade é utilizada. Se a unidade for utilizada novamente dentro desse intervalo, o temporizador é estendido para outros poucos segundos. Se a unidade não for utilizada neste intervalo, o motor é desligado.

3.8 RELÓGIOS

Os **relógios** (também chamados temporizadores) são essenciais para a operação de qualquer sistema de compartilhamento de tempo por várias razões. Eles mantêm a hora do dia e impedem que um processo monopolize a CPU, entre outras coisas. O software do relógio pode assumir a forma de um *driver* de dispositivo, mesmo que um relógio não seja um dispositivo de bloco como um disco, ou um dispositivo de caractere, como um terminal. Nosso exame dos relógios seguirá o mesmo padrão das seções anteriores: primeiro veremos o hardware e o software do relógio em geral e, então, veremos mais de perto como essas idéias são aplicadas no MINIX.

3.8.1 Hardware do Relógio

Dois tipos de relógios são comumente utilizados nos computadores e ambos são bem diferentes daqueles utilizados pelas pessoas. Os relógios mais simples são ligados em uma fonte de alimentação de 110 ou de 220 volts e causam uma interrupção a cada ciclo de voltagem, a 50 ou a 60Hz.

O outro tipo de relógio é construído a partir de três componentes: um oscilador de cristal, um contador e um registrador de armazenamento, como mostrado na Figura 3-23. Quando um pedaço de cristal de quartzo é cortado ade-

quadamente e montado sob tensão, pode-se fazê-lo gerar um sinal periódico de exatidão muito alta, geralmente no alcance de 5 a 100MHz, dependendo do cristal escolhido. Pelo menos um circuito desse tipo é normalmente encontrado em qualquer computador, oferecendo um sinal de sincronização para os vários circuitos do computador. Esse sinal alimenta o contador para fazê-lo contar para baixo até zero. Quando o contador atinge zero, ele causa uma interrupção da CPU.

Relógios programáveis geralmente têm vários modos de operação. No **modo *one-shot***, quando o relógio é iniciado, ele copia o valor do registrador de armazenamento no contador e, então, decrementa o contador em cada pulso do cristal. Quando o contador atinge zero, ele causa uma interrupção e pára até que seja explicitamente iniciado novamente pelo software. No **modo de onda quadrada**, depois de atingir zero e causar a interrupção, o registrador de armazenamento automaticamente é copiado no contador, e o processo inteiro é repetido indefinidamente. Essas interrupções periódicas são chamadas **tiques do relógio**.

A vantagem do relógio programável é que a freqüência de suas interrupções pode ser controlada por software. Se um cristal de 1MHz é utilizado, então, o contador pulsa a cada microssegundo. Com registradores de 16 bits, as interrupções podem ser programadas para ocorrer em intervalos de 1 microssegundo a 65,536 milissegundos. Chips de relógio programáveis normalmente contêm dois ou três relógios independentemente programáveis e também têm muitas outras opções (p. ex., contar para cima em vez de para baixo, interrupções desativadas e outras).

Para impedir que o tempo atual seja perdido quando a energia do computador é desligada, a maioria dos computadores tem um relógio de backup alimentado por bateria, implementado com o tipo de circuitos de baixo consumo utilizado em relógios digitais. O relógio de bateria pode ser lido na inicialização. Se o relógio de backup não estiver presente, o software pode solicitar ao usuário a data e a hora atuais. Há também um protocolo-padrão para um sistema em rede obter o tempo atual de um *host* remoto. Em qualquer caso, o tempo, então, é traduzido no número de tiques de relógio desde 12 A.M. **Universal Coordinated Time (UTC)** (antigamente conhecido como Hora Média de Greenwich) em 1° de janeiro de 1970, como o UNIX e o MINIX fazem, ou desde alguma outra marca. Em cada tique de relógio, o tempo real é incrementado por uma contagem. Normalmente programas utilitários são oferecidos para configurar manualmente o relógio do sistema e o relógio de backup e sincronizar os dois.

3.8.2 Software do Relógio

Tudo que o hardware de relógio faz é gerar interrupções a intervalos conhecidos. Tudo mais que envolve tempo deve ser feito pelo software, o *driver* do relógio. Os deveres exatos do *driver* de relógio variam entre sistemas operacionais, mas normalmente incluem a maioria dos seguintes:

1. Manter a hora do dia.
2. Impedir que processos executem por mais tempo do que lhes foi dado.
3. Contabilizar a utilização da CPU.
4. Tratar a chamada de sistema ALARM feita por processos de usuário.
5. Oferecer temporizadores *watchdog** ("cão de guarda") para partes do próprio sistema.
6. Traçar perfis, monitorar e reunir estatísticas.

A primeira função do relógio, manter a hora do dia (também chamada de **tempo real**), não é difícil. Isso exige apenas incrementar um contador a cada tique de relógio, como mencionado antes. A única coisa a monitorar é o número de bits no contador de hora do dia. Com um relógio de 60Hz, um contador de 32 bits irá estourar somente após mais de dois anos. Naturalmente o sistema não pode armazenar o tempo real como o número de tiques desde 1° de janeiro de 1970 em 32 bits.

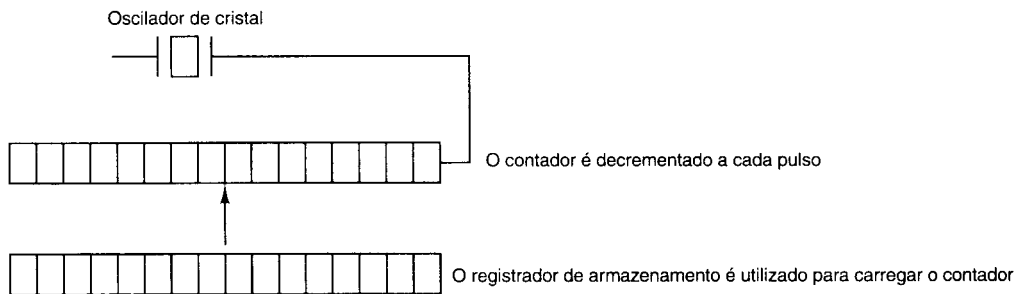



Figura 3-23 Um relógio programável.

*N. de R. *Watchdog Timer*: é um circuito acrescentado ao equipamento para gerar uma interrupção quando este não receber um RESET em um tempo determinado. Serve para tomar uma atitude no caso da CPU se perder, por exemplo, por motivo de ruído (ou BUG de software).

Três abordagens podem ser adotadas para resolver esse problema. A primeira é utilizar um contador de 64 bits, embora isso torne a manutenção do contador mais cara uma vez que tem de ser feita muitas vezes por segundo. A segunda é manter o tempo do dia em segundos, em vez de em tiques, utilizando um contador subsidiário para contar tiques até que um segundo inteiro tenha acumulado. Como 2^{32} segundos é de mais de 136 anos, esse método funcionará bem até o século XXII.

A terceira abordagem é contar em tiques, mas fazer isso em relação ao tempo em que o sistema é inicializado, em vez de relativo a um momento externo fixo. Quando o relógio de backup é lido ou o usuário digita o tempo real, o tempo de inicialização de sistema é calculado a partir do valor da hora atual do dia e armazenado na memória de forma conveniente. Mais tarde, quando a hora do dia for solicitada, a hora do dia armazenada é adicionada ao contador para obter a hora atual. Todas as três abordagens são mostradas na Figura 3-24.

A segunda função do relógio é impedir que processos executem por muito tempo. Sempre que um processo é iniciado, o agendador deve iniciar um contador para o valor do quantum desse processo em tiques de relógio. Em cada interrupção de relógio, o *driver* de relógio decrementa o contador do quantum por 1. Quando atinge zero, o tal *driver* de relógio chama o agendador para configurar outro processo.

A terceira função do relógio é fazer contabilidade da CPU. A maneira mais precisa para tanto é iniciar um segundo temporizador, distinto do temporizador principal do sistema, sempre que um processo é iniciado. Quando esse processo é parado, o temporizador pode ser lido para informar por quanto tempo o processo executou. Para fazer as coisas corretamente, o segundo temporizador deve ser salvo quando uma interrupção ocorre e restaurado depois.

Uma maneira menos precisa, mas muito mais simples, de fazer a contabilidade é manter um ponteiro para a entrada da tabela de processos para o processo atualmente em execução em uma variável global. Em cada tique de relógio, um campo na entrada do processo atual é incrementado. Assim, cada tique de relógio é contabilizado para o processo que executa no momento do tique. Um problema menor com essa estratégia é que, se muitas interrupções ocorrem durante a execução de um processo, ele é ainda cobrado por um tique inteiro, mesmo que não tenha feito muito trabalho. Uma contabilidade adequada para a CPU durante uma interrupção é muito cara e nunca é feita.

No MINIX e em muitos outros sistemas, um processo pode solicitar que o sistema operacional forneça-lhe um aviso depois de um certo intervalo. O aviso é normalmente um sinal, uma interrupção, uma mensagem ou algo semelhante. Uma aplicação que exige esse tipo de aviso é uma rede em que um pacote não-confirmado dentro de um certo intervalo de tempo deve ser retransmitido. Outra aplicação é instrução auxiliada por computador, na qual se um aluno não oferecer uma resposta dentro de um certo tempo a resposta lhe é informada.

Se o *driver* de relógio tivesse relógios suficientes, ele poderia definir um relógio separado para cada solicitação. Não sendo esse o caso, ele deve simular múltiplos relógios virtuais com um único relógio físico. Uma maneira de fazer isso é manter uma tabela em que o tempo de sinalização para todos os temporizadores pendentes é mantido, assim como uma variável que dá o tempo do próximo. Sempre que a hora do dia é atualizada, o *driver* verifica se o próximo sinal ocorreu. Se ocorreu, a próxima ocorrência é buscada na tabela.

Se muitos sinais são esperados, é mais eficiente simular múltiplos relógios encadeando todas as solicitações de relógios pendentes, classificadas segundo a hora, em uma lista encadeada, como mostrado na Figura 3-25. Cada entrada na lista informa quantos tiques de relógio após o anterior devem ser esperados antes de causar um sinal. Neste exemplo, os sinais são pendentes para 4203, 4207, 4213, 4215 e 4216.

Na Figura 3-25, a próxima interrupção ocorre em três tiques. Em cada tique, *Próximo sinal* é decrementado. Quando chega a 0, o sinal correspondente ao primeiro item na lista é causado e esse item é removido da lista. Então, *Próximo sinal* é configurado como o valor da entrada agora no topo da lista, neste exemplo, 4.

Note que durante uma interrupção de relógio, o *driver* de relógio tem várias coisas a fazer — incrementar o tempo real, decrementar o quantum e verificar para 0, fazer contabilidade da CPU e decrementar o contador do alar-

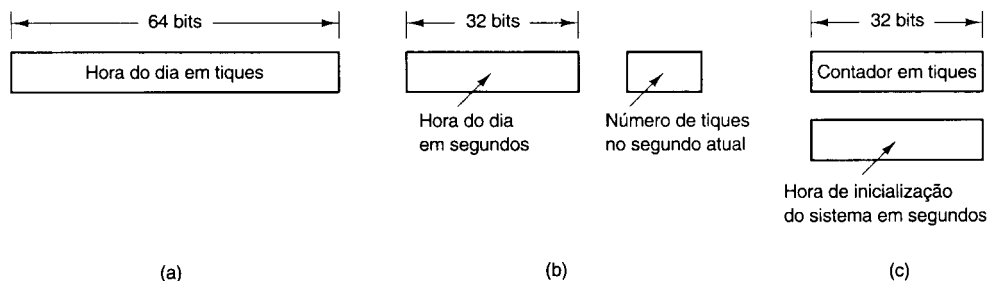



Figura 3-24 Três maneiras de manter a hora do dia.

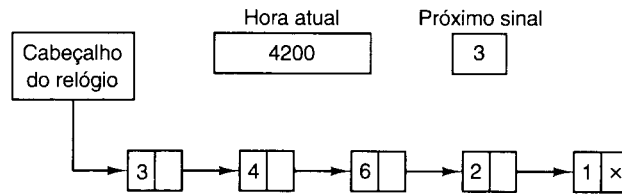



Figura 3-25 Simulando múltiplos temporizadores com um único relógio.

me. Entretanto, cada uma dessas operações foi cuidadosamente organizada para ser muito rápida porque precisa ser repetida muitas vezes por segundo.

Partes do sistema operacional também precisam de temporizadores. Esses são chamados **temporizadores *watchdog*** ("cão de guarda"). Ao estudar o *driver* de disco rígido, vimos que uma chamada de *wakeup* é agendada cada vez que um comando é enviado à controladora de disco; então, uma tentativa de recuperação pode ser feita se o comando falha completamente. Também mencionamos que os *drivers* de disquete precisam esperar o motor do disco atingir uma certa velocidade e devem desligar o motor se nenhuma atividade ocorrer em um certo tempo. Algumas impressoras com um cabeçote móvel de impressão podem imprimir em 120 caracteres/s (8.3ms/caractere), mas não podem retornar o cabeçote de impressão à margem esquerda em 8,3ms, assim o *driver* de terminal deve aguardar depois que um retorno de carro é digitado.

O mecanismo utilizado pelo *driver* de relógio para tratar temporizadores *watchdog* é o mesmo que para sinais de usuário. A única diferença é que quando um temporizador dispara, em vez de causar um sinal, o *driver* de relógio chama um procedimento fornecido pelo processo chamador. O procedimento é parte do código do processo chamador, mas como todos os *drivers* estão no mesmo espaço de endereço, o *driver* de relógio pode chamá-lo de qualquer jeito. O procedimento chamado pode fazer o que for necessário, até causar uma interrupção, embora dentro do *kernel* interrupções sejam freqüentemente inconvenientes e sinais não existam. Essa é a razão pela qual o mecanismo *watchdog* é oferecido.

A última coisa em nossa lista é traçar perfis. Alguns sistemas operacionais oferecem um mecanismo por meio do qual um programa de usuário pode fazer o sistema construir um histograma de seu contador de programa, de modo que ele pode ver onde está gastando seu tempo. Quando traçar perfis é uma possibilidade, em cada tique o *driver* verifica se o processo atual está tendo seu perfil traçado e, se estiver, ele calcula o número *bin* (um intervalo de endereços) correspondente ao contador de programa atual. Então, ele incrementa esse *bin* por um. Esse mecanismo também pode ser utilizado para traçar um perfil do próprio sistema.

3.8.3 Visão Geral do *Driver* de Relógio no MINIX

O *driver* de relógio do MINIX está contido no arquivo *clock.c*. A tarefa de relógio aceita estes seis tipos de mensagem, com os parâmetros mostrados:

1. HARD_INT
2. GET_UPTIME
3. GET_TIME
4. SET_TIME (novo tempo em segundos)
5. SET_ALARM (número de processo, procedimento a chamar, retardo)
6. SET_SYN_AL (número de processo, retardo)

HARD_INT é a mensagem enviada para o *driver* quando uma interrupção de relógio ocorre e há trabalho a fazer, como quando um alarme deve ser enviado ou um processo executou por muito tempo.

GET_UPTIME é utilizada para obter o tempo em tiques, desde a hora de inicialização, *GET_TIME* retorna o tempo real atual como o número de segundos passados desde as 12h de 1° de janeiro de 1970, e *SET_TIME* configura o tempo real. Ele somente pode ser chamado pelo superusuário.

Interno ao *driver* de relógio, o tempo é monitorado utilizando-se o método da Figura 3.24 (c). Quando o tempo é configurado, o *driver* calcula quando o sistema foi inicializado. Ele pode fazer essa computação porque tem o tempo real atual e também sabe por quantos tiques o sistema executou. O sistema armazena o tempo real da inicialização em uma variável. Mais tarde, quando *GET_TIME* é chamada, ela converte o valor atual do contador de tique para segundos e adiciona ao tempo da inicialização armazenado.

SET_ALARM permite que um processo configure um temporizador que dispara em um número especificado de tiques de relógio. Quando um processo de usuário faz uma chamada ALARM, ele envia uma mensagem para o gerenciador de memória, que, então, envia essa mensagem para o *driver* de relógio. Quando o alarme dispara, o *driver* de relógio envia uma mensagem para o gerenciador de memória, que, então, cuida de fazer o sinal acontecer.

SET_ALARM também é utilizada por tarefas que precisam iniciar um temporizador *watchdog*. Quando o tempo-

rizador dispara, o procedimento fornecido simplesmente é chamado. O *driver* de relógio não tem nenhum conhecimento do que o procedimento faz.

SET_SYN_AL é semelhante a *SET_ALARM*, mas é utilizada por um **alarme síncrono.** Um alarme síncrono envia uma mensagem para um processo, em vez de gerar um sinal ou chamar um procedimento. A tarefa do alarme síncrono envolve despachar mensagens para os processos que as solicitam. Alarmes síncronos serão discutidos detalhadamente mais tarde.

A tarefa de relógio não utiliza nenhuma estrutura de dados importante, mas há diversas variáveis utilizadas para monitorar o tempo. Somente uma é uma variável global, *lost_ticks*, definida em *glo.h* (linha 5031). Essa variável é fornecida para ser utilizada por qualquer *driver* que possa ser adicionado ao MINIX no futuro e que talvez desative as interrupções por um tempo suficientemente longo que um ou mais tiques de relógio possam ser perdidos. Atualmente ela não é utilizada, mas se um *driver* desse tipo fosse escrito o programador poderia fazer com que *lost_ticks* fosse incrementado para compensar o tempo durante o qual as interrupções de relógio foram inibidas.

Obviamente, as interrupções de relógio ocorrem muito freqüentemente e seu rápido tratamento é importante. O MINIX consegue isso fazendo uma quantidade mínima de processamento na maioria das interrupções de relógio. Ao receber uma interrupção, o manipulador configura uma variável local, *ticks*, como *lost_ticks* + 1 e, então, utiliza essa quantidade para atualizar os tempos de contabilidade e *pending_ticks* (linha 11079) e redefine *lost_ticks* para zero. *Pending_ticks* é uma variável *PRIVATE*, declarada fora de todas as definições de função, mas conhecida apenas pelas funções definidas em *clock.c*. Outra variável *PRIVATE*, *sched_ticks*, é decrementada em cada tique para acompanhar o tempo de execução. O manipulador de interrupções envia uma mensagem à tarefa de relógio somente se um alarme está vencido ou um quantum de execução foi utilizado. Esse esquema resulta no manipulador de interrupções, retornando quase imediatamente na maioria das interrupções.

Quando a tarefa de relógio recebe qualquer mensagem, ela adiciona *pending_ticks* à variável *realtime* (linha 11067) e, então, zera *pending_ticks*. *Realtime*, junto com a variável *boot_time* (linha 11068), permite que a hora atual do dia seja calculada. Ambas são variáveis *PRIVATE*; então, a única maneira de qualquer outra parte do sistema obter o tempo é enviando uma mensagem à tarefa de relógio. Embora em qualquer instante *realtime* possa estar incorreto, esse mecanismo assegura que ele seja sempre exato quando necessário. Se seu relógio está correto quando você o vê, importa se ele está incorreto quando você não está olhando?

Para manipular alarmes, *next_alarm* registra o tempo em que o próximo sinal ou chamada de *watchdog* pode acontecer. O *driver* precisa ser cuidadoso aqui, porque o processo que solicita o sinal pode sair ou pode ser eliminado antes de o sinal acontecer. Quando é hora do sinal, uma verificação é feita para ver se ainda é necessário. Se não for necessário, não será executado.

A cada processo de usuário permite-se apenas um temporizador. Executar uma chamada ALARM enquanto o temporizador ainda está executando cancela o primeiro temporizador. Portanto, uma maneira conveniente de armazenar os temporizadores é reservar uma palavra na entrada da tabela de processos para o temporizador de cada processo, se houver um. Para tarefas, a função a ser chamada também deve ser armazenada em algum lugar; então, uma matriz, *watch_dog*, foi oferecida para essa finalidade. Uma matriz semelhante, *syn_table*, armazena ainalizadores para indicar a cada processo se ele deve obter um alarme síncrono.

A lógica geral do *driver* de relógio segue o mesmo padrão dos *drivers* de disco. O programa principal é um laço sem fim que recebe mensagens, despacha de acordo com o tipo de mensagem e, então, envia uma resposta (exceto para *CLOCK_TICK*). Cada tipo de mensagem é tratado por um procedimento separado, seguindo nossa convenção de atribuir nomes a todos os procedimentos chamados a partir do laço principal como *do_xxx*, onde *xxx* é diferente para cada um, naturalmente. A propósito, muitos *linkeditors*, infelizmente, truncam nomes de procedimento para sete ou para oito caracteres; assim, os nomes *do_set_time* e *do_set_alarm* estão potencialmente em conflito. O último foi renomeado para *do_setalarm*. Esse problema ocorre em todo o MINIX e normalmente é resolvido quebrando-se alguns nomes.

A Tarefa de Alarme Síncrono

Há uma segunda tarefa a ser discutida nesta seção, a **tarefa de alarme síncrono**. Um alarme síncrono é semelhante a um alarme, mas em vez de enviar um sinal para chamar uma função *watchdog* quando o período de tempo limite expira, a tarefa de alarme síncrono envia uma mensagem. Um sinal pode chegar ou uma tarefa *watchdog* pode ser chamada sem qualquer relação com que parte de uma tarefa está executando, portanto, alarmes desse tipo são **assíncronos.** Em contraposição, uma mensagem é recebida somente quando o receptor executou uma chamada *receive*.

O mecanismo de alarme síncrono foi adicionado ao MINIX para suportar o servidor de rede, que, como o gerenciador de memória e o servidor de arquivos, executa como um processo separado. Com freqüência, precisa-se de um limite no tempo em que um processo pode ser bloqueado enquanto espera por entradas. Por exemplo, em uma rede, a falha em obter a confirmação de um pacote de dados dentro de um período definido é provavelmente devida a uma falha de transmissão. Um servidor de rede pode configurar um alarme síncrono antes de tentar receber uma mensagem e bloquear. Como o alarme síncrono é entregue como uma mensagem, ele acabará desbloqueando o servidor se nenhuma mensagem for recebida da rede. Ao obter qualquer mensagem, o servidor deve primeiro redefinir o

alarme. Então, examinando o tipo ou a origem da mensagem, ele pode determinar se um pacote chegou ou se foi desbloqueado por um tempo limite. Se for o último caso, o servidor pode tentar uma recuperação, normalmente reenviando o último pacote não-confirmado.

Um alarme síncrono é mais rápido que um alarme enviado utilizando um sinal, que exige várias mensagens e uma quantidade considerável de processamento. Uma função de *watchdog* é rápida, mas é útil somente para tarefas compiladas no mesmo espaço de endereço que a tarefa de relógio. Quando um processo está esperando uma mensagem, um alarme síncrono é mais apropriado e mais simples que sinais ou funções *watchdog* e facilmente é tratado com um pequeno processamento adicional.

O Manipulador de Interrupções do Relógio

Como descrito anteriormente, quando uma interrupção de relógio ocorre, *realtime* não é atualizado imediatamente. A rotina de serviço de interrupção mantém o contador *pending_ticks* e faz trabalhos simples como contabilizar o tique atual para um processo e decrementar o temporizador do quantum. Uma mensagem é enviada para a tarefa de relógio somente quando trabalhos mais complicados devem ser feitos. Isso é algo de acordo com o ideal do MINIX de as tarefas comunicarem-se totalmente por mensagens, mas também é uma concessão à realidade de que servir tiques de relógio consome tempo de CPU. Estimou-se que em uma máquina lenta fazer isso dessa maneira resulta em um aumento de 15% na velocidade do sistema em relação a uma implementação que envia uma mensagem à tarefa de relógio em cada interrupção de relógio.

Sincronização em Milissegundos

Como outra concessão à realidade, são apresentadas em *clock.c* algumas rotinas que oferecem precisão de milissegundos. Retardos de até um milissegundo são necessários para vários dispositivos de E/S. Não há maneira prática de fazer isso utilizando alarmes e a interface de passagem de mensagens. As funções aqui destinam-se a serem chamadas diretamente pelas tarefas. A técnica utilizada é a mais antiga e simples técnica de E/S: acesso direto. O contador que é utilizado para gerar a interrupção de relógio é lido diretamente, o mais rápido possível, e a contagem é convertida em milissegundos. O chamador faz isso repetidamente até que o tempo desejado tenha passado.

Resumo dos Serviços de Relógio

A Figura 3-26 resume os vários serviços fornecidos por *clock.c*. Há várias maneiras de acessar o relógio e várias maneiras de as solicitações serem atendidas. Alguns serviços estão disponíveis para qualquer processo, e os resultados são retornados em uma mensagem.

O tempo de funcionamento pode ser obtido por uma chamada de função a partir do *kernel* ou por uma tarefa para evitar o *overhead* de uma mensagem. Um alarme pode ser solicitado por um processo de usuário, sendo o resultado final disso um sinal ou, para uma tarefa, a ativação de uma função *watchdog*. Nenhum desses mecanismos pode ser utilizado por um processo de servidor, mas um servidor pode solicitar um alarme síncrono. Uma tarefa ou o *kernel* pode solicitar um retardo que utiliza a função *milli_delay* ou pode incorporar chamadas para *milli_elapsed* em uma rotina de acesso direto, por exemplo, enquanto espera a entrada de uma porta.

3.8.4 Implementação do *Driver* de Relógio no MINIX

Quando o MINIX inicia, todos os *drivers* são chamados. A maioria deles apenas tenta obter uma mensagem e bloqueia. O *driver* de relógio, *clock_task* (linha 11098), também faz isso, mas primeiro chama *init_clock* para inicializar a freqüência programável do relógio para 60Hz. Quando qualquer mensagem é recebida, ele adiciona *pending_ticks* a *realtime* e, então, zera *pending_ticks* antes de fazer qualquer outra coisa. Essa operação potencialmente poderia entrar em conflito com uma interrupção de relógio; assim chamadas *lock* e *unlock* são utilizadas para prevenir uma condição de corrida (linhas 11115 a 11118). De resto, o laço principal do *driver* de relógio é essencialmente o mesmo que os outros *drivers*: uma mensagem é recebida, uma função que realiza o trabalho solicitado é chamada, e uma mensagem de resposta é enviada.

Do_clocktick (linha 11140) não é chamada em cada tique do relógio; então, seu nome não é uma descrição exata de sua função. Ela é chamada quando o manipulador de interrupções determinou que pode haver algo importante a fazer. Primeiro, uma verificação é feita para ver se um temporizador de sinal ou o *watchdog* disparou. Se isso aconteceu, todas as entradas de alarme na tabela de processos são inspecionadas. Como os tiques não são processados individualmente, vários alarmes podem disparar em uma passagem pela tabela. Também é possível que o processo que deveria obter o próximo alarme já tenha encerrado. Quando é localizado um processo cujo alarme é menor que o tempo atual, mas não zero, a entrada da matriz *watch_dog* correspondente a tal processo é verificada. Na linguagem de programação C, um valor numérico também tem um valor lógico, assim o teste na linha 11161 retorna *TRUE* se um endereço válido estiver armazenado na entrada em *watch_dog* e a função correspondente é chamada indiretamente na linha 11163. Se um **ponteiro nulo** for encontrado (representado em C pelo valor zero), o teste é avaliado como *FALSE* e *cause_sig* é chamada para enviar um sinal SIGALRM. A entrada em *watch_dog* também é utilizada quando um alarme síncrono é necessário. Nesse caso, o endereço armazenado é o endereço de *cause_alarm*, em vez de o endereço de uma função *wa-*

Serviço	Acesso	Resposta	Clientes
Gettime	Chamada de sistema	Mensagem	Qualquer processo
Uptime	Chamada de sistema	Mensagem	Qualquer processo
Uptime	Chamada de função	Valor da função	Kernel ou tarefa
Alarme	Chamada de sistema	Sinal	Qualquer processo
Alarme	Chamada de sistema	Ativação de Watchdog	Tarefa
Alarme síncrono	Chamada de sistema	Mensagem	Processo de servidor
Milli_delay	Chamada de função	Espera ativa	Kernel ou tarefa
Milli_elapsed	Chamada de função	Valor da função	Kernel ou tarefa

Figura 3-26 O código de relógio suporta diversos serviços relacionados com tempo.

tchdog pertencente a uma tarefa em particular. Para enviar um sinal, poderíamos ter armazenado o endereço de *cause_sig*, mas, então, precisaríamos ter escrito *cause_sig* de maneira diferente, para não esperar nenhum argumento e obter o número do processo de destino de uma variável global. Alternativamente, poderíamos ter exigido que todos os processos *watchdog* esperassem um argumento que eles não precisam.

Discutiremos *cause_sig* quando tratarmos da tarefa de sistema em uma seção adiante. Seu trabalho é enviar uma mensagem para o gerenciador de memória. Isto requer uma verificação de se o gerenciador de memória atualmente está esperando uma mensagem. Em caso afirmativo, envia uma mensagem informando sobre o alarme. Se o gerenciador de memória está ocupado, uma nota é feita para informá-lo na primeira oportunidade.

Ao fazer o laço pela tabela de processos inspecionando o valor de *p_alarm* para cada processo, *next_alarm* é atualizado. Antes de iniciar o laço é configurado para um número muito grande (linha 11151) e, então, para cada processo cujo valor do alarme é diferente de zero depois de enviar alarmes ou sinais, uma comparação é feita entre o alarme do processo e *next_alarm,* o qual é configurado para o menor valor (linhas 11171 e 11172).

Depois de processar alarmes, *do_clocktick* prossegue para ver se é hora de agendar outro processo. O quantum de execução é mantido na variável *PRIVATE sched_ticks*, que normalmente é decrementada pelo manipulador de interrupções de relógio em cada tique de relógio. Entretanto, nesses tiques, quando *do_clocktick* é ativado, não é decrementada pelo manipulador, permitindo que o próprio *do_clocktick* faça isso e teste quanto a um resultado zero na linha 11178. *Sched_ticks* não é redefinida sempre que um novo processo é agendado (porque o sistema de arquivos e o gerenciador de memória têm permissão para executar até sua conclusão). Em vez disso, ela é redefinida depois de cada *SCHED_RATE* tiques. A comparação na linha 11179 é para assegurar que o processo atual realmente executou pelo menos um tique completo do agendador antes de tomar a CPU dele.

O próximo procedimento, *do_getuptime* (linha 11189), é somente uma linha; ele coloca o valor atual de *realtime* (o número de tiques desde a inicialização) no campo adequado na mensagem a ser retornada. Qualquer processo pode obter o tempo passado dessa maneira, mas o *overhead* da mensagem é um preço alto a exigir das tarefas, portanto, é oferecida uma função relacionada, *get_uptime* (linha 11200), que pode ser chamada diretamente pelas tarefas. Como não é chamada via uma mensagem à tarefa de relógio, ela mesma precisa adicionar os tiques pendentes ao *realtime* atual. *Lock* e *unlock* são necessários aqui para impedir que uma interrupção de relógio ocorra enquanto *pending_ticks* está sendo acessado.

Para obter o tempo real atual, *do_get_time* (linha 11219) calcula o tempo real atual a partir de *realtime* e *boot_time* (o tempo de inicialização do sistema em segundos). *Do_set_time* (linha 11230) é seu complemento. Ele calcula um novo valor para *boot_time* com base no tempo real atual dado e no número de tiques desde a inicialização.

Os procedimentos *do_setalarm* (linha 11242) e *do_setsyn_alrm* (linha 11269) são tão semelhantes que os discutiremos juntos. Ambos extraem os parâmetros que especificam o processo a ser sinalizado e o tempo de espera da mensagem. *Do_setalarm* também extrai uma função a chamar (linha 11257), embora algumas linhas mais adiante substituam esse valor por um ponteiro nulo se o processo de destino for um processo de usuário e não uma tarefa. Já vimos como esse ponteiro é mais tarde testado em *do_clocktick* para determinar se o destino deve obter um sinal ou uma chamada para um *watchdog*. O tempo restante para o alarme (em segundos) também é calculado pelas duas funções e configurado na mensagem de retorno. Ambos, então, chamam *common_setalarm* para encerrar. No caso da chamada *do_setsyn_alarm*, o parâmetro de função passado para *common_setalarm* é sempre *cause_alarm*.

Commom_setalarm (linha 11291) termina o trabalho iniciado por qualquer uma das duas funções que acabamos de discutir. Então, ela armazena a hora do alarme

na tabela de processos e o ponteiro para o procedimento *watchdog* (que também pode ser um ponteiro para *cause_alarm* ou um ponteiro nulo) na matriz *watch_dog*. Então, ele varre a tabela de processos inteira para encontrar o próximo alarme, assim como é feito por *do_clocktick*.

Cause_alarm (linha 11318) é simples; ele define como *TRUE* uma entrada na matriz *syn_table* correspondente ao destino do alarme síncrono. Se a tarefa de alarme síncrono não estiver ativa, é enviada uma mensagem para acordá-la.

Implementação da Tarefa de Alarme Síncrono

A tarefa de alarme síncrono, *syn_alarm_task* (linha 11333), segue o modelo básico de todas as tarefas. Inicia e, então, entra em um laço interminável em que recebe e envia mensagens. A inicialização consiste em declarar-se viva configurando a variável *syn_al_alive* como *TRUE* e, então, declarando que não tem nada a fazer colocando todas as entradas em *syn_table* como *FALSE*. Há uma entrada em *syn_table* para cada entrada na tabela de processos. Ela começa seu laço externo declarando concluído seu trabalho e, então, entra em um laço interno onde verifica todas as entradas em *syn_table*. Se localiza uma entrada que indica que um alarme síncrono é esperado, ela redefine a entrada, envia uma mensagem do tipo *CLOCK_INT* para o processo apropriado e declara seu trabalho não-concluído. Na parte inferior do seu laço externo, ela não faz pausa para esperar qualquer nova mensagem a menos que seu sinalizador *work_done* seja configurado. Uma nova mensagem não é necessária para informar que há mais trabalho a fazer, uma vez que *cause_alarm* escreve diretamente em *syn_table*. Uma mensagem é necessária somente para acordá-la depois que ela executar todo o trabalho. O efeito é que ela faz um ciclo muito rapidamente, contanto que haja alarmes a serem entregues.

De fato, essa tarefa não é utilizada pela versão de distribuição do MINIX. Entretanto, se você compilar o MINIX adicionando suporte de rede, ela será utilizada pelo servidor de rede, o qual precisa exatamente desse tipo de mecanismo para impor limites de tempo rápidos se pacotes não forem recebidos quando esperados. Além da necessidade de velocidade, não se pode enviar um sinal a um servidor, uma vez que os servidores devem executar eternamente, e a ação-padrão da maioria dos sinais é eliminar o processo de destino.

A Implementação do Manipulador de Interrupções de Relógio

O projeto do manipulador de interrupções de relógio é um compromisso entre fazer muito pouco (assim o tempo de processamento será minimizado) e fazer o suficiente para tornar infreqüentes as caras ativações da tarefa de relógio. Ele muda algumas variáveis e testa algumas outras. *Clock_handler* (linha 11374) inicia fazendo a contabilidade do sistema. O MINIX monitora tanto o tempo do usuário como o do sistema. O tempo do usuário é cobrado de um processo se ele estiver executando quando ocorre um tique do relógio. O tempo de sistema é cobrado se o sistema de arquivos ou o gerenciador de memória estiver executando. A variável *bill_ptr* sempre aponta para o último processo de usuário agendado (os dois servidores não contam). A cobrança é feita nas linhas 11447 e 11448. Depois que a cobrança termina, a variável mais importante mantida por *clock_handler*, *pending_ticks*, é incrementada (linha 11450). O tempo real deve ser conhecido para testar se *clock_handler* deve acordar *tty* ou enviar uma mensagem à tarefa de relógio; porém, realmente atualizar o próprio *realtime* é caro, porque essa operação deve ser feita utilizando bloqueios. Para evitar isso, o manipulador calcula sua própria versão do tempo real na variável local *now*. Há uma pequena chance de que o resultado seja incorreto de vez em quando, mas as conseqüências de tal erro não seriam sérias.

O restante do trabalho do manipulador depende de vários testes. O terminal e a impressora precisam ser acordados de vez em quando. *Tty_timeout* é uma variável global, mantida pela tarefa de terminal, que armazena o valor de quando *tty* deve ser acordada. Para a impressora, diversas variáveis que são *PRIVATE* dentro do módulo da impressora precisam ser verificadas e testadas na chamada a *pr_restart*, o qual retorna rapidamente mesmo no pior caso de a impressora estar desligada. Nas linhas 11455 a 11458, é feito um teste que ativa a tarefa de relógio se um alarme estiver vencido ou se é tempo de agendar outra tarefa. O último teste é complexo, um AND lógico de três testes mais simples. O código

```
interrupt(CLOCK);
```

na linha 11459 resulta em uma mensagem *HARD_INT* para a tarefa de relógio.

Ao discutir *do_clocktick*, notamos que ela decrementa *sched_ticks* e testa quanto a zero para ver se o quantum de execução expirou. Testar se *sched_ticks* é igual a um é parte do teste complexo que mencionamos acima; se a tarefa de relógio não é ativada, ainda é necessário decrementar *sched_ticks* dentro do manipulador de interrupções e, se atingir zero, redefinir o quantum. Se isso ocorrer, é hora também de anotar que o processo atual estava ativo no início do novo quantum; isso é feito pela alocação do valor atual de *bill_ptr* para *prev_ptr* na linha 11466.

Os Utilitários de Tempo

Por fim, *clock.c* contêm algumas funções que oferecem vários tipos de suporte. Muitas dessas funções são específicas do hardware e precisarão ser substituídas ao portar o MINIX para hardware não-Intel. Descreveremos apenas a função dessas, sem entrar em seus detalhes internos.

Init_clock (linha 11474) é chamada pela tarefa do temporizador quando executa pela primeira vez. Ela configura o modo e o retardo de tempo do chip do temporizador

para produzir interrupções de tique de relógio 60 vezes por segundo. Apesar do fato de que a "velocidade de CPU" que se vê em anúncios para PCs aumentou de 4,77MHz no IBM PC original para mais de 200MHz nos sistemas modernos, a constante *TIMER_COUNT*, utilizada para iniciar o temporizador, é a mesma independente do modelo de PC em que o MINIX está executando. Todo PC compatível com IBM, independentemente da rapidez com que seu processador trabalha, oferece um sinal de 14,3MHz para utilização por vários dispositivos que precisam de uma referência de tempo. As linhas seriais de comunicações e a exibição de vídeo também precisam dessa referência de sincronização.

O complemento de *init_clock* é *clock_stop* (linha 11489). Não é realmente necessário, mas é um concessão ao fato de que os usuários do MINIX podem querer iniciar outro sistema operacional às vezes. Ele simplesmente redefine os parâmetros do chip do temporizador para o modo-padrão de operação que o MS-DOS e outros sistemas operacionais podem esperar o ROM BIOS oferecer quando eles iniciam pela primeira vez.

Milli_delay (linha 11502) é oferecido para utilização por qualquer tarefa que precisa de retardos muito curtos. Ela é escrita em C sem quaisquer referências específicas de hardware, mas utiliza uma técnica que talvez se espere encontrar somente em uma rotina de baixo nível de linguagem *assembly*. Ela inicia um contador em zero e, então, rapidamente o consulta continuamente até que um valor desejado seja alcançado. No Capítulo 2, dissemos que essa técnica de espera ativa deveria, em geral, ser evitada, mas as necessidades da implementação podem exigir exceções às regras gerais. A inicialização do contador é feita pela próxima função, *milli_start* (linha 11516), que simplesmente zera duas variáveis. A consulta é feita chamando a última função, *milli_elapsed* (linha 11529), que acessa o hardware do temporizador. O contador que é examinado é o mesmo utilizado para contar para baixo tiques de relógio e pode sofrer *underflow** e ser redefinido para seu valor máximo antes de o retardo desejado estar completo. *Milli_elapsed* corrige isso.

3.9 TERMINAIS

Cada computador de propósito geral tem um ou mais terminais utilizados para comunicar-se com ele. Os terminais são apresentados em um número extremamente grande de formas diferentes. Cabe ao *driver* de terminal esconder todas essas diferenças, de modo que a parte independente de dispositivo do sistema operacional e os programas de usuário não precisem ser escritos para cada tipo de terminal. Nas próximas seções, seguiremos nossa abordagem-padrão de primeiro discutir o hardware e o software terminal em geral e, então, discutir o software do MINIX.

*N. de R. Na linguagem C não há verificação de limite nas operações aritméticas; logo quando se subtrai 1 de uma variável inteira com valor zero ela passa a representar o valor máximo da faixa de representação de seu tipo. Isso caracteriza um *underflow* ou "estouro" inferior da faixa de representação.

3.9.1 Hardware de Terminal

Do ponto de vista do sistema operacional, os terminais podem ser divididos em três categorias amplas baseadas em como o sistema operacional comunica-se com eles. A primeira categoria consiste em terminais mapeados em memória, que se compõem de um teclado e de um dispositivo de exibição, ambos conectados como parte do hardware do computador. A segunda categoria são terminais que interfaceiam via uma linha comunicação serial utilizando o RS-232 padrão, freqüentemente por meio de um modem. A terceira categoria consiste em terminais que são conectados ao computador via uma rede. Essa taxonomia é mostrada na Figura 3-27.

Terminais Mapeados em Memória

A primeira categoria ampla de terminais mostrada na Figura 3-27 compõe-se de terminais mapeados em memória, que são parte integrante dos próprios computadores. Os terminais mapeados em memória são interfaceados via uma memória especial chamada **RAM de vídeo**, que forma parte do espaço de endereçamento do computador e é endereçada pela CPU da mesma maneira que o restante da memória (ver a Figura 3-28).

Também na placa de RAM de vídeo está um chip chamado **controladora de vídeo.** Esse chip puxa códigos de caractere da RAM de vídeo e gera o sinal utilizado para orientar o monitor. O monitor gera um feixe de elétrons que varre a tela horizontalmente, pintando linhas nele. Geralmente a tela tem 480 a 1024 linhas de cima para baixo, com 640 a 1.200 pontos por linha. Esses pontos são chamados ***pixels***. O sinal da controladora de vídeo modula o feixe de elétrons, determinando se um dado pixel será claro ou escuro. Os monitores coloridos têm três feixes, para vermelho, verde e azul, que são independentemente modulados.

Um dispositivo de exibição monocromática simples pode ajustar cada caractere em uma caixa de 9 *pixels* de largura por 14 *pixels* de altura (incluindo o espaço entre caracteres) e tem 25 linhas de 80 caracteres. A tela, então, teria 350 linhas de varredura com 720 *pixels* cada uma. Cada um desses quadros é redesenhando 45 a 70 vezes por segundo. A controladora de vídeo poderia ser projetada para buscar os primeiros 80 caracteres da RAM de vídeo, gerar 14 linhas de varredura, buscar os próximos 80 da RAM de vídeo, gerar as 14 linhas de varredura seguintes e assim por diante. De fato, a maioria busca cada caractere uma vez por linha de varredura para eliminar a necessidade de buffers na controladora. Os padrões de 9 por 14 bits para os caracteres são mantidos em uma ROM utilizada pela controladora de vídeo. (A RAM também pode ser utilizada para suportar fontes personalizadas.) A ROM é endereçada por um endereço de 12 bits; 8 bits do código de caractere e 4

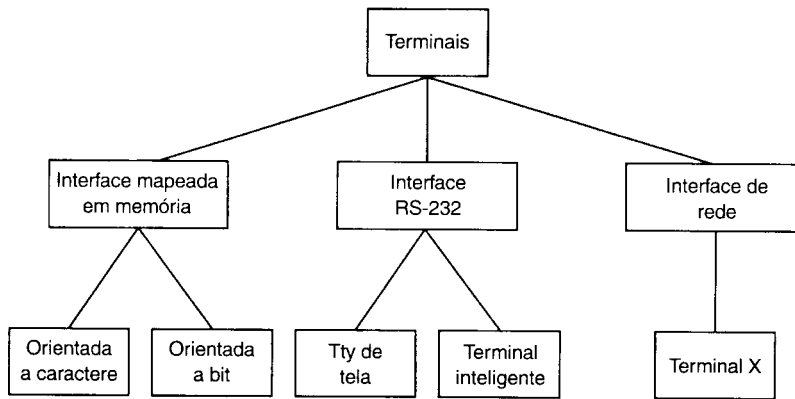



Figura 3-27 Tipos de terminal.

bits especificam uma linha de varredura. Os 8 bits em cada byte da ROM controlam 8 *pixels*; o 9º *pixel* entre caracteres está sempre em branco. Portanto, 14 × 80 = 1120 referências de memória à RAM de vídeo são necessárias por linha de texto na tela. O mesmo número de referências é feito para a ROM do gerador de caracteres.

O IBM PC tem vários modos para a tela. No mais simples, ele utiliza um vídeo mapeado por caracteres para o console. Na Figura 3-29(a) vemos uma parte da RAM de vídeo. Cada caractere na tela da Figura 3-29(b) ocupa dois caracteres na RAM. O caractere de ordem inferior é o código ASCII para o caractere a ser exibido. O caractere de ordem superior é o byte de atributo, que é utilizado para especificar a cor, o vídeo reverso, a intermitência e assim por diante. A tela completa de 25 por 80 caracteres requer 4.000 bytes de RAM de vídeo neste modo.

Os terminais de mapa de bits utilizam o mesmo princípio, exceto que cada *pixel* na tela é controlado individualmente. Na configuração mais simples, para um vídeo monocromático, cada *pixel* tem um bit correspondente na RAM de vídeo. No outro extremo, cada *pixel* é representado por um número de 24 bits, com 8 bits para cada vermelho, verde e azul. Um vídeo colorido de 768 × 1024 com 24 bits por *pixel* requer 2MB de RAM apenas para armazenar a imagem.

Com um vídeo mapeado em memória, o teclado está completamente separado da tela. Ele pode ser interfaceado via uma porta serial ou paralela. Em cada ação de teclado, a CPU é interrompida, e o *driver* de teclado extrai o caractere digitado lendo uma porta de E/S.

No IBM PC, o teclado contém um microprocessador embutido que se comunica por meio de uma porta serial especializada com um chip de controladora na placa-mãe. Uma interrupção é gerada sempre que uma tecla é pressionada e também quando uma tecla é liberada. Além disso, tudo que o hardware de teclado oferece é o número da tecla, não o código ASCII. Quando a tecla *A* é pressionada, o código da tecla (30) é colocado em um registrador de E/S. Cabe ao *driver* determinar se é caixa baixa (minúscula), caixa alta (maiúscula), CTRL-A, ALT-A, CTRL-ALT-A ou alguma outra combinação. Como o *driver* pode informar quais teclas foram pressionadas, mas ainda não liberadas (p. ex., *shift*), ele tem informações suficientes para fazer o trabalho. Embora essa interface de teclado coloque todo o peso sobre o software, ela é extremamente flexível. Por exemplo, programas de usuário podem interessar-se se um

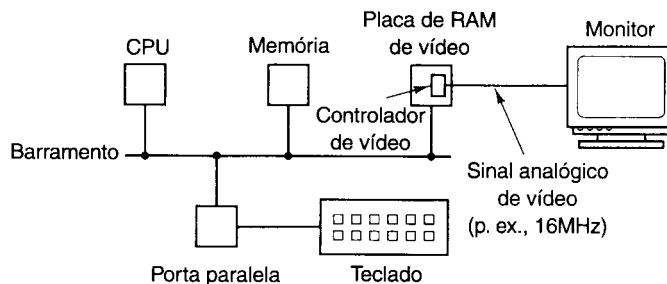



Figura 3-28 Terminais de memória mapeada gravam diretamente na RAM de vídeo.

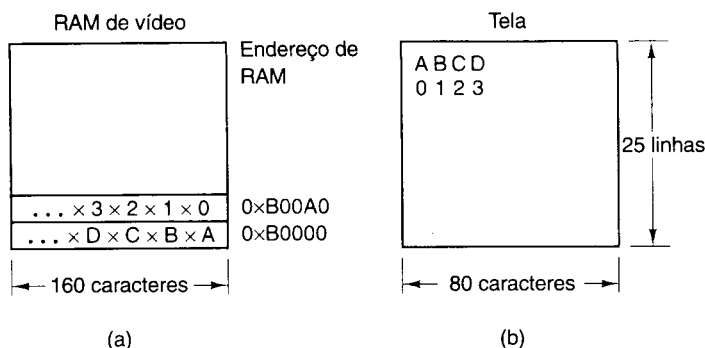



Figura 3-29 (a) Uma imagem de RAM de vídeo para o monitor monocromático IBM. (b) A tela correspondente. Os ×s são bytes de atributo.

algarismo recém-digitado veio da fila superior de teclas ou do teclado numérico. A princípio, o *driver* pode oferecer essas informações.

Terminais RS-232

Terminais RS-232 são dispositivos que contêm um teclado e um monitor que se comunicam utilizando uma interface serial, um bit por vez (ver Figura 3-30). Esses terminais utilizam um conector de 9 pinos ou de 25 pinos, dos quais um pino é utilizado para transmitir dados, um é para obter dados e um é terra. Os outros pinos são para várias funções de controle, a maioria das quais não é utilizada. Para enviar um caractere para um terminal RS-232, o computador deve transmiti-lo 1 bit por vez, prefixado por um bit de partida e seguido por 1 ou 2 bits de parada para delimitar o caractere. Um bit de paridade que oferece detecção rudimentar de erro também pode ser inserido precedendo os bits de parada, embora isso comumente seja requerido somente para comunicação com sistemas *mainframe*. Taxas comuns de transmissão de dados são 9.600, 19.200 e 38.400bps. Terminais RS-232 são comumente utilizados para comunicação com um computador remoto, utilizando um modem e uma linha de telefone.

Uma vez que tanto os computadores como os terminais trabalham internamente com caracteres inteiros mas devem comunicar-se por uma linha serial um bit por vez, foram desenvolvidos chips para fazer conversões caractere para serial e serial para caractere. Eles são chamados **UARTs** (*Universal Asynchronous Receiver Transmitters*). Os UARTs são ligados ao computador conectando-se placas de interface RS-232 no barramento como ilustrado na Figura 3-31. Os terminais RS-232 estão gradualmente desaparecendo, sendo substituídos por PCs e por terminais X, mas eles ainda são encontrados em antigos sistemas de *mainframe* especialmente em bancos, em reservas de passagens aéreas e aplicativos semelhantes.

Para imprimir um caractere, o *driver* de terminal grava o caractere na placa de interface, onde ele é *bufferizado* e, então, é enviado por uma linha serial um bit por vez pelo UART. Mesmo a 38.400bps, leva mais de 250 microssegundos para enviar um caractere. Como resultado dessa taxa de transmissão lenta, o *driver* geralmente produz um caractere para a placa RS-232 e bloqueia, esperando a interrupção gerada pela interface quando o caractere for transmitido, e o UART for capaz de aceitar outro caractere. O UART pode enviar e receber caracteres simultaneamente, como seu nome indica. Uma interrupção também é ge-

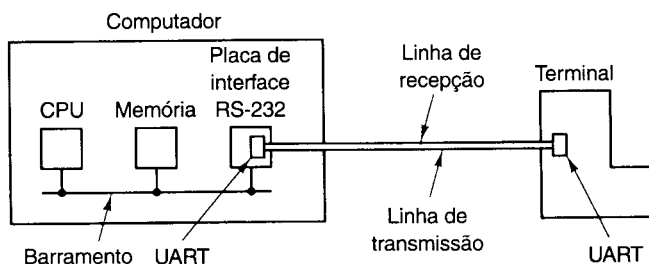



Figura 3-30 Um terminal RS-232 comunica-se com um computador por uma linha de comunicação, um bit por vez. O computador e o terminal são completamente independentes.

rada quando um caractere é recebido e normalmente um número pequeno de caracteres de entrada pode ser *bufferizado*. O *driver* de terminal deve verificar um registrador quando uma interrupção é recebida para determinar a causa da interrupção. Algumas placas de interface têm uma CPU e memória e podem tratar múltiplas linhas, assumindo grande parte da carga de E/S da CPU principal.

Terminais RS-232 podem ser subdivididos em categorias, conforme já mencionado. Os mais simples eram terminais de impressão. Os caracteres digitados no teclado eram transmitidos para o computador. Os caracteres enviados pelo computador eram impressos em papel. Esses terminais estão obsoletos e raramente são vistos hoje em dia.

Terminais *burros* de CRT trabalham da mesma maneira, exceto que utilizam uma tela em vez de papel. Eles freqüentemente são chamados "*ttys* de tela" ("*glass ttys*") porque são funcionalmente os mesmos que os *ttys* de impressão. (O termo "*tty*" é uma abreviação de Teletype®, uma antiga empresa que foi pioneira no negócio de terminais de computadores; *tty* acabou tornando-se sinônimo de qualquer terminal.) Os *ttys* de tela também estão obsoletos.

Terminais inteligentes de CRT são de fato miniaturas de computadores especializados. Eles têm uma CPU e memória e contêm software, normalmente em ROM. Do ponto de vista do sistema operacional, a diferença principal entre um *tty* de tela e um terminal inteligente é que o último entende certas seqüências de escape. Por exemplo, enviando-se o caractere ASCII ESC (033), seguido por vários outros caracteres, pode-se mover o cursor para qualquer posição na tela, inserir texto no meio da tela, etc.

Terminais X

A última palavra em terminais inteligentes é um terminal que contém uma CPU tão poderosa quanto o computador principal, junto com vários megabytes de memória, um teclado e um mouse. Um terminal comum desse tipo é o **terminal X**, que roda no *X Window* do M.I.T. Em geral, os terminais *X Window System* conversam com o computador principal sobre uma rede Ethernet.

Um terminal X é um computador que executa o software X. Alguns produtos são dedicados para executar somente X; outros são computadores de propósito geral que simplesmente executam como um programa entre muitos outros. De qualquer maneira, um terminal X tem uma grande tela de mapa de bits normalmente com resolução de 960 × 1.200 ou melhor, em escala de cinza, branco e preto ou colorido, um teclado completo e um mouse, normalmente com três botões.

O programa dentro do terminal X que recebe a entrada do teclado ou do mouse e aceita comandos de um computador remoto é chamado **servidor X.** Ele se comunica pela rede com **clientes X** que rodam em algum *host* remoto. Pode parecer estranho ter o servidor X dentro do terminal e os clientes no *host* remoto, mas o trabalho do servidor X é exibir bits; então, faz sentido estar próximo do usuário. A organização cliente-servidor é mostrada na Figura 3-31.

A tela do terminal X contém algumas janelas, cada uma na forma de uma grade retangular de *pixels*. Cada janela normalmente tem uma barra de título na parte superior, uma barra de rolagem à esquerda e uma caixa de redimensionamento no canto superior direito. Um dos clientes

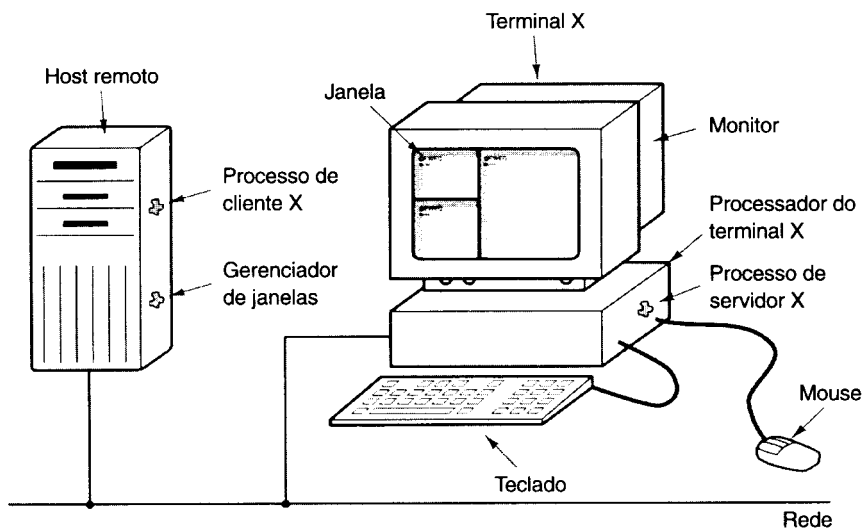



Figura 3-31 Os clientes e servidores no *X Window System* do M.I.T.

X é um programa chamado **gerenciador de janelas**. Seu trabalho é controlar a criação, a exclusão e o movimento das janelas na tela. Para gerenciar janelas, ele envia comandos para o servidor X instruindo o que fazer. Esses comandos incluem desenhar um ponto, desenhar uma linha, desenhar um retângulo, desenhar um polígono, preencher um retângulo, preencher um polígono e assim por diante.

O trabalho do servidor X é coordenar entrada de mouse, teclado e clientes X e atualizar o monitor de maneira correspondente. Ele precisa monitorar qual janela está atualmente selecionada (onde o ponteiro do mouse está), para saber para qual cliente enviar qualquer nova entrada de teclado.

3.9.2 Software de Terminal

O teclado e o monitor são dispositivos quase independentes, assim os trataremos separadamente aqui. (Eles não são precisamente independentes, já que caracteres digitados devem ser exibidos na tela.) No MINIX, os *drivers* de teclado e de tela são partes da mesma tarefa; em outros sistemas eles podem dividir-se em *drivers* distintos.

O Software de Entrada

O trabalho básico do *driver* de teclado é reunir entradas do teclado e passar para os programas de usuário quando elas forem lidas do terminal. Duas filosofias podem ser adotadas para o *driver*. Na primeira, o trabalho do *driver* é simplesmente aceitar entrada e passá-la para cima inalterada. Um programa que lê do terminal obtém uma seqüência de códigos ASCII brutos. (Fornecer aos programas de usuário os números de tecla é muito primitivo, além de ser grandemente dependente da máquina.)

Essa filosofia atende bem às necessidades dos editores de tela sofisticados como o *emacs*, que permite que o usuário associe uma ação arbitrária a qualquer caractere ou seqüência de caracteres. Entretanto, isso significa que se o usuário digitar *dsta* em vez de *data* e, então, corrigir o erro digitando três *backspace* e *ata*, seguido por um retorno de carro, o programa de usuário receberá todos os 11 códigos ASCII digitados.

A maioria dos programas não exige tantos detalhes assim. Eles apenas querem a entrada corrigida, não a seqüência exata de como foi produzida. Essa observação conduz à segunda filosofia: o *driver* trata toda edição entre linhas e entrega somente linhas corrigidas para os programas de usuário. A primeira filosofia é baseada em caractere; a segunda é baseada em linha. Originalmente elas eram referidas como **modo bruto** (*raw mode*) e **modo processado** (*cooked mode*), respectivamente. O padrão POSIX utiliza o termo menos pitoresco **modo canônico** para descrever o modo baseado em linha. Na maioria dos sistemas do modo canônico significa uma configuração bem-definida. O **modo não-canônico** é equivalente ao modo bruto, embora muitos detalhes do comportamento do terminal possam ser alterados. Os sistemas compatíveis com POSIX oferecem várias funções de biblioteca que suportam selecionar qualquer um dos modos e alterar muitos aspectos da configuração do terminal. No MINIX, a chamada de sistema IOCTL suporta essas funções.

A primeira tarefa do *driver* de teclado é completar caracteres. Se cada pressionamento de tecla causar uma interrupção, o *driver* pode obter o caractere durante a interrupção. Se as interrupções são transformadas em mensagens pelo software de baixo nível, é possível colocar o caractere recentemente obtido na mensagem. Alternativamente, ele pode ser colocado em um pequeno buffer na memória e a mensagem utilizada para informar o *driver* de que algo chegou. A última abordagem é realmente mais segura se uma mensagem puder ser enviada somente para um processo em espera e houver alguma chance de o *driver* de teclado ainda estar ocupado com o caractere anterior.

Uma vez que o *driver* recebeu o caractere, ele deve começar a processá-lo. Se o teclado entrega números de tecla em vez dos códigos de caractere utilizados pelo software aplicativo, então, o *driver* deve converter os códigos utilizando uma tabela. Nem todos os sistemas compatíveis com o padrão IBM utilizam a numeração padrão de teclas; então, se o *driver* quiser suportar essas máquinas, ele deverá mapear teclados diferentes com tabelas diferentes. Uma abordagem simples é compilar uma tabela que mapeia os códigos fornecidos pelo teclado para os códigos ASCII (*American Standard Code for Information Interchange*) no *driver* de teclado, mas isso é insatisfatório para usuários de idiomas que não o inglês. Os teclados são organizados diferentemente em países diferentes, e o conjunto de caracteres ASCII não é adequado nem mesmo para a maioria das pessoas no hemisfério Ocidental, onde os idiomas espanhol, português e francês precisam de caracteres acentuados e marcas de pontuação não-utilizadas no inglês. Para responder à necessidade de flexibilidade nos leiautes de teclado para diferentes idiomas, muitos sistemas operacionais oferecem **mapas de teclado** ou **páginas de código** carregáveis, que tornam possível escolher o mapeamento entre códigos de teclado e códigos entregues para o aplicativo, seja quando o sistema é inicializado, seja mais tarde.

Se o terminal estiver no modo canônico (processado), os caracteres devem ser armazenados até que uma linha inteira seja acumulada, pois o usuário pode depois decidir apagar parte dela. Mesmo que o terminal esteja no modo bruto, o programa pode ainda não ter solicitado a entrada, assim os caracteres devem ser *bufferizados* para permitir armazenar teclas digitadas. (Projetistas de sistema que não permitem que os usuários digitem muito adiante, isto é, que não oferecem um buffer de teclado razoável, deveriam ser mergulhados em um balde de piche e cobertos de penas, ou, pior ainda, forçados a utilizar seu próprio sistema.)

Duas abordagens para *bufferização* de caracteres são comuns. Na primeira, o *driver* contém um conjunto central de buffers, cada buffer armazenando talvez 10 caracteres. Associada com cada terminal está uma estrutura de dados, que contém, entre outros itens, um ponteiro para a cadeia de buffers para a entrada coletada desse terminal. À

medida que mais caracteres são digitados, mais buffers são adquiridos e incluídos na cadeia. Quando os caracteres são passados para um programa de usuário, os buffers são removidos e devolvidos para o conjunto central.

A outra abordagem é fazer a *bufferização* diretamente na própria estrutura de dados do terminal, sem nenhum conjunto de buffers central. Uma vez que é comum os usuários digitarem um comando que levará algum tempo (digamos, uma compilação) e, então, digitar algumas linhas adiante, por segurança o *driver* deveria atribuir algo como uns 200 caracteres por terminal. Em um sistema de compartilhamento de tempo de grande escala com 100 terminais, alocar 20K o tempo todo para o *type ahead* é evidentemente exagerado, então, um conjunto central buffers com espaço para talvez 5K é provavelmente suficiente. Por outro lado, um buffer dedicado por terminal torna o *driver* mais simples (não há gerenciamento de lista encadeada) e seria preferido em computadores pessoais com somente um ou dois terminais. A Figura 3-32 mostra a diferença entre esses dois métodos.

Embora o teclado e o monitor sejam dispositivos lógicos separados, muitos usuários cresceram acostumados a ver os caracteres que eles acabam de digitar aparecer na tela. Alguns terminais (antigos) exibem automaticamente (em hardware) o que se acabou de digitar, o que não somente é um incômodo quando senhas estão sendo inseridas como também limita grandemente a flexibilidade dos editores sofisticados e de outros programas. Felizmente, os terminais mais modernos não exibem nada quando teclas são digitadas. Portanto cabe ao software exibir a entrada. Esse processo é chamado **ecoamento**.

O ecoamento é complicado pelo fato de que um programa pode estar escrevendo na tela enquanto o usuário está digitando. No mínimo, o *driver* de teclado tem de imaginar onde colocar a nova entrada sem ser sobrescrito pela saída de programa.

O ecoamento também fica complicado quando mais de 80 caracteres são digitados em um terminal com linhas de 80 caracteres. Dependendo do aplicativo, a quebra de linha pode ser apropriada. Alguns *drivers* simplesmente truncam as linhas para 80 caracteres jogando fora todos os caracteres além de coluna 80.

Outro problema é o tratamento de tabulação. A maioria dos terminais tem uma tecla de tabulação, mas poucos podem tratar tabulação na saída. Cabe ao *driver* calcular onde o cursor está atualmente localizado, levar em conta tanto a saída dos programas como a saída de ecoamento e calcular o número adequado de espaços a ser ecoado.

Agora chegamos ao problema de equivalência de dispositivo. Logicamente, no fim de uma linha de texto, queremos um retorno de carro, mover de volta o cursor para a coluna 1 e uma quebra de linha, para avançar para a próxima linha. Exigir que os usuários digitem os dois comandos no fim de cada linha não daria certo (embora alguns terminais tenha uma tecla que gera ambos, com 50% de chance de fazer isso na ordem que o software quer). Cabe ao *driver* converter a entrada para o formato interno padrão utilizado pelo sistema operacional.

Se a forma-padrão é simplesmente armazenar uma quebra de linha (a convenção do MINIX), então, os retornos de carro transformam-se em quebras de linha. Se o formato interno é armazenar ambos, então, o *driver* deve gerar uma quebra de linha quando receber um retorno de carro e retorno de carro quando receber uma quebra de linha. Independentemente da convenção interna, o terminal pode solicitar que tanto uma quebra de linha como um retorno de carro sejam ecoados a fim de atualizar a tela adequadamente. Uma vez que um computador de gran-

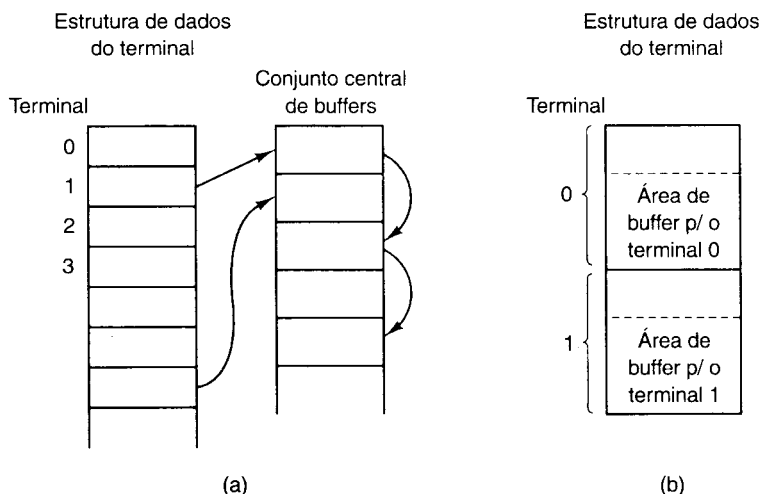



Figura 3-32 (a) Conjunto central de buffers. (b) Buffer dedicado para cada terminal.

de porte pode ter uma grande variedade de terminais diferentes conectados a ele, cabe ao *driver* de teclado receber todas as combinações diferentes de quebra de linha e retorno de carro convertidos ao padrão interno do sistema e organizar para todo ecoamento ser feito corretamente.

Um problema relacionado é a sincronização de retorno de carro e as quebras de linha. Em alguns terminais, pode levar mais tempo para exibir um retorno de carro ou quebra de linha que uma letra ou um número. Se o microprocessador dentro do terminal precisar realmente copiar um bloco grande de texto para conseguir fazer a tela rolar, então, as quebras de linha podem ser lentas. Se um cabeçote de impressão mecânico precisa ser retornado para a margem esquerda do papel, os retornos de carro podem ser lentos. Em ambos os casos, cabe ao *driver* inserir os **caracteres de preenchimento** (caracteres nulos fictícios) no fluxo de saída ou simplesmente interromper a saída por tempo suficiente para o terminal alcançá-lo. A quantidade de tempo de retardo freqüentemente está relacionada com a velocidade do terminal, por exemplo, a 4800bps ou mais lento, talvez nenhum retardo seja necessário, mas a 9600bps ou velocidade mais alta, talvez os caracteres de preenchimento sejam necessários. Os terminais com tabulações de hardware, especialmente aqueles de *hardcopy*, também podem solicitar um retardo depois de uma tabulação.

Ao operar em modo canônico, vários caracteres de entrada têm significados especiais. A Figura 3-33 mostra todos os caracteres especiais necessários para o POSIX e os adicionais reconhecidos pelo MINIX. Os padrões são que todos os caracteres de controle não devem gerar conflito com a entrada de texto nem com os códigos utilizados por programas, mas todos exceto os dois últimos podem ser alterados utilizando o comando *stty*, se desejado. Versões mais antigas do UNIX utilizavam diferentes padrões para muitos desses.

O caractere *ERASE* permite que o usuário apague o caractere que acabou de digitar. No MINX ele é o *backspace* (CTRL-H). Ele não é adicionado à fila de caracteres, mas, em vez disso, remove o caractere anterior da fila. Ele deve ser ecoado como uma seqüência de três caracteres, *backspace*, espaço e *backspace*, a fim de remover o caractere anterior da tela. Se o caractere anterior era uma tabulação, a operação de apagá-lo requer monitorar onde o cursor estava antes da tabulação. Na maioria dos sistemas, usar *backspace* apagará apenas os caracteres na linha atual. Não apagará retorno de carro e voltará à linha anterior.

Quando o usuário nota um erro no início da linha sendo digitada, é com freqüência conveniente apagar a linha inteira e iniciar novamente. O caractere *KILL* (no MINIX o CRTL-U) apaga a linha inteira. O MINIX faz a linha apagada desaparecer da tela, mas alguns sistemas ecoam-na com mais um retorno de carro e com uma quebra de linha porque alguns usuários querem ver a linha antiga. Portanto, ecoar *KILL* é uma questão de gosto. Como com *ERASE* normalmente não é possível voltar além da linha atual. Quando um bloco de caracteres é eliminado, ele pode ou não levar ao problema de o *driver* retornar buffers para o *pool*, se um é utilizado.

Eventualmente os caracteres *ERASE* ou *KILL* devem ser inseridos como dados normais. O caractere *LNEXT* serve como um **caractere de escape.** No MINIX, Ctrl é o padrão. Como um exemplo, os sistemas UNIX mais antigos freqüentemente utilizavam o sinal @ para *KILL*, mas o sistema de correio da Internet utiliza endereços na forma *linda@cs.washington.edu*. Alguém que se sinta mais confortável com as convenções mais antigas pode redefinir *KILL* como

Caractere	Nome no POSIX	Comentário
CRTL-D	EOF	Fim de arquivo
	EOL	Fim de linha (não definido)
CRTL-H	ERASE	Retroceder um caractere
DEL	INTR	Interrompe o processo (SIGINT)
CRTL-U	KILL	Apaga a linha inteira que esta sendo digitada
CRTL-\	QUIT	Força dump de núcleo (SIGQUIT)
CRTL-Z	SUSP	Suspende (ignorado pelo MINIX)
CRTL-Q	START	Inicia a saída
CRTL-S	STOP	Pára a saída
CRTL-R	REPRINT	Reexibe a entrada (extensão do MINIX)
CRTL–V	LNEXT	Literal seguinte (extensão do MINIX)
CRTL-O	DISCARD	Descarta a saída (extensão do MINIX)
CRTL-M	CR	Retorno de carro (inalterável)
CRTL-J	NL	Quebra de linha (inalterável)

Figura 3-33 Caracteres que recebem tratamento especial no modo canônico.

@, mas, então, é necessário inserir um sinal @ literalmente para endereçar correio eletrônico. Isso pode ser feito digitando-se CTRL-V @.O próprio CTRL-V pode ser inserido literalmente digitando-se CTRL-V CTRL-V. Depois de ver um CTRL-V, o *driver* configura um sinalizador avisando que o próximo caractere é isento de processamento especial. O caractere *LNEXT* em si não entra na fila de caracteres.

Para permitir que os usuários impeçam a rolagem da imagem da tela para fora do campo de visão, códigos de controle são oferecidos para congelar a tela e reiniciá-la mais tarde. No MINIX, esses códigos são *STOP* (CTRL-S) e *START* (CTRL-Q), respectivamente. Eles não são armazenados, mas utilizados para ligar e para desligar um sinalizador na estrutura de dados do terminal. Sempre que uma saída é tentada, o sinalizador é inspecionado. Se estiver ligado, nenhuma saída ocorre. Normalmente, o ecoamento também é suprimido junto com a saída do programa.

Com freqüência, é necessário eliminar um programa que está sendo depurado. Os caracteres *INTR* (DEL) e QUIT (CTRL-\) podem ser utilizados para esse propósito. No MINIX, DEL envia o sinal de SIGINT para todos os processos iniciados a partir do terminal. Implementar DEL é bem difícil. A parte difícil é levar as informações do *driver* para a parte do sistema que trata sinais, que, afinal de contas, não solicitou tais informações. O CTRL-\ é semelhante a DEL, exceto que envia o sinal de SIGQUIT, que força um *dump* de núcleo se não capturado ou ignorado. Quando qualquer uma dessas teclas é pressionada, o *driver* deve ecoar um retorno de carro e uma quebra de linha e descartar toda entrada acumulada para permitir uma inicialização atualizada. O valor-padrão para *INTR* é, com freqüência, CTRL-C em vez de DEL, pois muitos programas utilizam DEL intercambiavelmente com o *backspace* para edição.

Outro caractere especial é *EOF* (CTRL-D), que no MINIX faz com que qualquer solicitação pendente para o terminal seja atendida com qualquer coisa que esteja disponível no buffer, mesmo que o buffer esteja vazio. Digitar CTRL-D no início de uma linha faz com que o programa obtenha um leitura de 0 bytes, que convencionalmente é interpretado como fim de arquivo e faz com que a maioria dos programas comporte-se da mesma maneira como se comportaria ao ver o fim de arquivo em um arquivo de entrada.

Alguns *drivers* de terminal oferecem muito mais recursos de edição entre linhas do que esboçamos aqui. Eles têm caracteres especiais de controle para apagar uma palavra, pular caracteres ou palavras para trás ou para frente para ir para o começo ou para o fim da linha sendo digitada, etc. Adicionar todas essas funções ao *driver* de terminal torna-o muito maior e, ademais, tudo isso é desperdiçado quando se utilizam editores de tela que trabalham em modo bruto no final das contas.

Para permitir que os programas controlem parâmetros de terminal, o POSIX requer que várias funções estejam disponíveis na biblioteca-padrão, das quais as mais importantes são *tcgetattr* e *tcsetattr*. *Tcgetattr* recupera uma cópia da estrutura mostrada na Figura 3-34, a estrutura *termios*, que contém todas as informações necessárias para mudar caracteres especiais, para configurar modos e para modificar outras características de um terminal. Um programa pode examinar as configurações atuais e modificá-las conforme desejado. *Tcsetattr*, então, grava a estrutura de volta à tarefa de terminal.

O POSIX não especifica se seus requisitos devem ser implementados por funções de biblioteca ou por chamadas de sistema. O MINIX oferece uma chamada de sistema, IOCTL, chamada por

```
ioctl(file_descriptor, request, argp);
```

que é utilizada para examinar e para modificar as configurações de muitos dispositivos de E/S. Essa chamada é utilizada para implementar as funções *tcgetattr* e *tcsetattr*. A variável *request* especifica se a estrutura *termios* é para ser lida ou gravada e, no último caso, se a solicitação deve ser atendida imediatamente ou adiada até que toda a saída atualmente enfileirada esteja completa. A variável *argp* é um ponteiro para uma estrutura *termios* no programa de chamada. Essa opção particular de comunicação entre o programa e o *driver* foi escolhida por sua compatibilidade com UNIX, não pela sua beleza inerente.

Algumas notas sobre a estrutura *termios* são necessárias. As quatro palavras de sinalização oferecem muita flexibilidade. Os bits individuais em *c_iflag* controlam várias maneiras como a entrada é tratada. Por exemplo, o bit *ICRNL* faz com que os caracteres *CR* sejam convertidos em *NL* na entrada. Esse sinalizador é ligado por padrão no MINIX. O *c_oflag* armazena bits que afetam o processamento da saída. Por exemplo, o bit *OPOST* ativa processamento de saída. Ele e o bit *ONLCR*, que faz com que os caracteres *NL* na saída sejam convertidos em uma seqüência *CR NL*, também são configurados por padrão no MINIX. O *c_cflag* é o

```
struct termios {
  tcflag_t c_iflag;        /* modos de entrada */
  tcflag_t c_oflag;        /* modos de saída */
  tcflag_t c_cflag;        /* modos de controle */
  tcflag_t c_lflag;        /* modos locais */
  speed_t c_ispeed;        /* taxa de entrada */
  speed_t c_ospeed;        /* taxa de saída */
  cc_t c_cc[NCCS];         /* caracteres de controle */
};
```

Figura 3-34 A estrutura *termios*. No MINIX *tc_flag_t* é um *short*, *speed_t* é um *int*, *cc_t* é um *char*.

sinalizador de controle. As configurações padrão para o MINIX permitem que uma linha receba caracteres de 8 bits e fazem com que um modem desligue se um usuário desconectar-se da linha. O *c_lflag* é o campo dos sinalizadores de modo local. Um bit, *ECHO*, ativa o ecoamento (isso pode ser desativado durante um *login* para oferecer segurança ao inserir-se uma senha). Seu bit mais importante é o *ICANON*, que ativa o modo canônico. Com *ICANON* desligado, há várias possibilidades. Se todas as outras configurações são deixadas em seus padrões, um modo idêntico ao tradicional **modo cbreak** é iniciado. Nesse modo, os caracteres são passados para o programa sem esperar uma linha completa, mas os caracteres *INTR*, *QUIT*, *START* e *STOP* mantêm seus efeitos. Entretanto, todos esses podem ser desativados redefinindo bits nos sinalizadores, produzindo o equivalente do modo bruto tradicional.

Os vários caracteres especiais que podem ser mudados, incluindo aqueles que são extensões do MINIX, são armazenados na matriz *c_cc*. Essa matriz também armazena dois parâmetros que são utilizados no modo não-canônico. A quantidade *MIN*, armazenada em *c_cc*[*VMIN*], especifica o número mínimo de caracteres que deve ser recebido para satisfazer a chamada READ. A quantidade TIME em *c_cc* [*VTIME*] define um limite de tempo para essas chamadas. *MIN* e *TIME* interagem como mostrado na Figura 3-35. Uma chamada que solicita *N* bytes é ilustrada. Com *TIME* = 0 e *MIN* = 1, o comportamento é semelhante ao modo bruto tradicional.

O Software de Saída

A saída é mais simples que a entrada, mas *drivers* para terminais RS-232 são radicalmente diferentes dos *drivers* para terminais de memória mapeada. O método que comumente é utilizado para terminais RS-232 é ter buffers de saída associados com cada terminal. Os buffers podem vir do mesmo *pool* que os buffers de entrada ou são dedicados, como a entrada. Quando os programas gravam no terminal, a saída é primeiro copiada para os buffers. De maneira semelhante, a saída de ecoamento também é copiada para os buffers. Depois que toda a saída foi copiada para os buffers (ou se os buffers estiverem cheios), é feita a saída do primeiro caractere, e o *driver* vai dormir. Quando a interrupção chega, é feita a saída do próximo caractere e assim por diante.

Com terminais mapeados em memória, é possível um esquema mais simples. Os caracteres a serem impressos são extraídos, um por vez, do espaço do usuário e colocados diretamente na RAM de vídeo. Com terminais RS-232, cada caractere a sair é simplesmente enviado pela linha para o terminal. Com memória mapeada, alguns caracteres exigem tratamento especial, entre eles, *backspace*, retorno de carro, quebra de linha e sinal sonoro (CTRL-G). Um *driver* para um terminal de memória mapeada deve monitorar em software a posição atual na RAM de vídeo, de modo que os caracteres imprimíveis possam ser colocados ali e a posição atual avançada. *Backspace*, retorno de carro e quebra de linha exigem que essa posição seja atualizada apropriadamente.

Em particular, quando uma quebra de linha é emitida na última linha, a tela deve ser rolada. Para ver como a rolagem funciona, veja a Figura 3-29. Se a controladora de vídeo sempre começasse a ler a RAM em 0xB0000, a única maneira de rolar a tela seria copiar 24 × 80 caracteres (cada caractere solicitando 2 bytes) de 0 x00B00A0 para 0xB0000, uma proposta que consome tempo.

Felizmente, o hardware normalmente oferece alguma ajuda aqui. A maioria das controladoras de vídeo contém um registrador que determina onde na RAM de vídeo deve-se começar a buscar bytes para a linha superior da tela. Configurando esse registrador para apontar para 0x00B00A0 em vez de 0xB0000, a linha que era previamente número dois move-se para o topo, e a tela inteira rola para cima uma linha. A única outra coisa que o *driver* deve fazer é copiar o que for necessário para a nova linha final. Quando a controladora de vídeo chegar ao topo da RAM, ela simplesmente continua a buscar bytes, começando no endereço mais baixo.

Outra questão com que o *driver* deve lidar em um terminal mapeado em memória é o posicionamento do cursor. Mais uma vez, o hardware geralmente oferece algum auxílio na forma de um registrador que informa onde o cursor deve ir parar. Por fim, há o problema do sinal sonoro que é emitido dando saída a uma onda senoidal ou quadrada para os alto-falantes, uma parte do computador bem separada da RAM de vídeo.

	TIME = 0	TIME > 0
MIN = 0	Retorna imediatamente com quaisquer coisas que estejam disponíveis, 0 a N bytes	O temporizador inicia imediatamente. Retorna com o primeiro byte inserido ou com 0 bytes depois do tempo limite
MIN > 0	Retorna com pelo menos MIN e até N bytes. Possível bloco indefinido.	O temporizador interbyte inicia após o primeiro byte. Retorna N bytes se atingir o tempo limite. Possível bloco indefinido.

Figura 3-35 *MIN* e *TIME* determinam quando uma chamada a *read* retorna em modo não-canônico. *N* é o número de bytes exigidos.

Vale notar que muitas das questões com que o *driver* de terminal defronta-se para um monitor mapeado em memória (rolar a tela, emitir um sinal sonoro e assim por diante) também são encaradas pelo microprocessador dentro de um terminal RS-232. Do ponto de vista do microprocessador, ele é o processador principal em um sistema com um monitor mapeado em memória.

Os editores de tela e muitos outros programas sofisticados precisam ser capazes de atualizar a tela de maneiras mais complexas que simplesmente rolar texto sobre o fundo do monitor. Para atendê-los, muitos *drivers* de terminal suportam uma variedade de seqüências de escape. Embora alguns terminais suportem conjuntos de seqüências de escape idiossincrásicos, é vantajoso ter um padrão para facilitar a adaptação do software de um sistema para outro. O *American National Standards Institute* (ANSI) definiu um conjunto de seqüências de escape-padrão, e o MINIX suporta um subconjunto das seqüências do ANSI, mostrado na Figura 3-36, que é adequado para muitas operações comuns. Quando o *driver* vê o caractere que inicia as seqüências de escape, ele ativa um sinalizador e espera até que o resto da seqüência de escape entre. Quando tudo tiver chegado, o *driver* deve executar a seqüência em software. Inserir e excluir texto exige mover blocos de caracteres pela da RAM de vídeo. O hardware não oferece qualquer ajuda exceto rolar e exibir o cursor.

3.9.3 Visão Geral do *Driver* de Terminal no MINIX

O *driver* de terminal é contido em quatro arquivos de C (seis se o suporte a RS-232 e a pseudoterminal forem ativados) e juntos eles constituem de longe o maior *driver* no MINIX. O tamanho do *driver* de terminal é em parte explicado notando que o *driver* trata o teclado e o monitor, cada um dos quais é um dispositivo complicado e com suas próprias particularidades, bem como dois outros tipos de terminal opcionais. Além disso, a maioria das pessoas surpreende-se ao descobrir que a E/S de terminal requer 30 vezes mais código que o agendador. (Essa sensação é reforçada vendo os numerosos livros sobre sistemas operacionais que dedicam 30 vezes mais espaço ao agendamento do que a toda a E/S combinada.)

O *driver* de terminal aceita sete tipos de mensagem:

1. Ler do terminal (a partir do sistema de arquivos em nome de um processo de usuário).
2. Gravar no terminal (a partir do sistema de arquivos em nome de um processo de usuário).
3. Configurar parâmetros de terminal para IOTCL (a partir do sistema de arquivos em nome de um processo de usuário).
4. E/S ocorrida durante o último tique do relógio (a partir da interrupção de relógio).
5. Cancelar a solicitação anterior (a partir do sistema de arquivos quando um sinal ocorre).
6. Abrir um dispositivo.
7. Fechar um dispositivo.

As mensagens para ler e gravar têm o mesmo formato que o mostrado na Figura 3-15, exceto que nenhum campo *POSITION* é necessário. Com um disco, o programa precisa especificar que bloco quer ler. Com um terminal, não há nenhuma escolha: o programa sempre recebe o próximo caractere digitado. Os terminais não suportam buscas.

As funções POSIX *tcgetattr* e *tcgetattr*, utilizadas para examinar e para modificar atributos de terminal (proprie-

Seqüência de escape	Significado
ESC [*n*A	Move para cima *n* linhas
ESC [*n*B	Move para baixo *n* linhas
ESC [*n*C	Move para a direita *n* espaços
ESC [ND	Move à esquerda *n* espaços
ESC [*m*;*n*H	Move cursor para (*m*,*n*)
ESC [*s* J	Limpa a tela do cursor (0 ao fim, 1 do início, 2 tudo)
ESC [*s* K	Limpa a linha do cursor (0 ao fim, 1 do início, 2 tudo)
ESC [*n* L	Insere *n* linhas no cursor
ESC [*n* M	Exclui *n* linhas no cursor
ESC [*n* P	Exclui *n* caracteres no cursor
ESC [*n* @	Insere *n* caracteres no cursor
ESC [*n* *m*	Ativa estilo de exibição (0=normal, 4=negrito, 5=intermitente, 7=inverso)
ESC M	Rola a tela para trás se o cursor está na linha superior

Figura 3-36 As seqüências de escape ANSI aceitas pelo *driver* de terminal na saída. ESC denota o caractere de escape ASCII (0x1B), e *n*, *m* e *s* são parâmetros numéricos opcionais.

dades), são suportadas pela chamada de sistema IOCTL. Uma boa prática de programação é utilizar essas funções e outras em *include/termios.h* e deixar para a biblioteca de C converter chamadas de biblioteca em chamadas de sistema IOCTL. Há, entretanto, algumas operações de controle necessárias para o MINIX que não são oferecidas no POSIX para, por exemplo, carregar um mapa de teclado alternativo; para essas operações o programador deve utilizar IOCTL explicitamente.

A mensagem enviada para o *driver* por uma chamada de sistema IOCTL contém um código de solicitação de função e um ponteiro. Para a função *tcsetattr*, uma chamada IOCTL é feita com um tipo de solicitação *TCSETS*, *TCSETSW* ou *TCSETSF* e um ponteiro para uma estrutura *termios* como mostrado na Figura 3-34. Todas essas chamadas substituem o conjunto de atributos atuais por um novo conjunto, sendo que as diferenças são que uma solicitação *TCSETS* tem efeito imediato, enquanto uma solicitação *TCSETSW* não tem efeito até que toda saída tenha sido transmitida e uma *TCSETSF* espera a saída terminar e descarta toda entrada que ainda não foi lida. *Tcgetattr* é traduzido em uma chamada IOCTL com um tipo de solicitação *TCGETS* e retorna para o processo chamador uma estrutura *termios* preenchida; assim, o estado atual de um dispositivo pode ser examinado. As chamadas IOCTL que não correspondem a funções definidas pelo POSIX, como a solicitação *KIOCSMAP*, utilizada para carregar um novo mapa de teclado, passam ponteiros para outros tipos de estruturas, neste caso para uma estrutura *keymap_t* que é uma estrutura de 1536 bytes (códigos de 16 bits para 128 teclas × 6 modificadores). A Figura 3-43 resume como as chamadas POSIX-padrão são convertidas em chamadas de sistema IOCTL.

O *driver* de terminal utiliza uma estrutura principal de dados, *tty_table*, que é uma matriz de estruturas *tty*, uma por terminal. Um PC-padrão tem apenas um teclado e um monitor, mas o MINIX pode suportar até oito terminais virtuais, dependendo da quantidade de memória na placa adaptadora do monitor. Isso permite que a pessoa no console conecte-se múltiplas vezes, troque a saída de monitor e a entrada de teclado de um "usuário" para outro. Com dois consoles virtuais, pressionar ALT-F2 seleciona o segundo, e ALT-F1 retorna o primeiro. ALT também pode ser utilizada com as teclas de seta. Além disso, linhas seriais podem suportar dois usuários em posições remotas, conectados por cabo RS-232 ou por modem, e os **pseudoterminais** podem suportar usuários conectados por uma rede. O *driver* foi escrito para facilitar o acréscimo de terminais adicionais. A configuração padrão ilustrada no código-fonte desse texto tem dois consoles virtuais, com linhas seriais e pseudoterminais desativados.

Cada estrutura *tty* em *tty_table* monitora tanto a entrada como a saída. Para a entrada, ela armazena uma fila de todos os caracteres que foram digitados, mas ainda não lidos pelo programa, as informações sobre solicitações para ler caracteres que ainda não foram recebidos e as informações de tempo limite, para que a entrada possa ser solicitada sem que a tarefa bloqueie permanentemente se nenhum caractere for digitado. Para a saída, ela armazena os parâmetros das solicitações de gravação que ainda não terminaram. Outros campos armazenam diversas variáveis gerais, como a estrutura *termios* discutida anteriormente, que afeta muitas propriedades tanto da entrada como da saída. Há também um campo na estrutura *tty* apontando para as informações que são necessárias para uma classe particular de dispositivos, mas não são necessárias na entrada *tty_table* para cada dispositivo. Por exemplo, a parte dependente do hardware do *driver* de console precisa da posição atual na tela e na RAM de vídeo e do byte de atributo atual para o monitor, mas essas informações não são necessárias para suportar uma linha RS-232. As estruturas de dados privadas para cada tipo de dispositivo são também onde os buffers que recebem entrada das rotinas de serviço de interrupções estão localizados. Dispositivos lentos, como o teclado, não precisam de buffers tão grandes quanto aqueles necessários para dispositivos velozes.

Entrada de Terminal

Para melhor entender como o *driver* funciona, vejamos primeiro como os caracteres digitados no terminal fazem seu caminho do sistema ao programa que os quer.

Quando um usuário conecta-se no console do sistema, um *shell* é criado para ele com */dev/console* como entrada-padrão, como saída-padrão e como erro padrão. O *shell* inicia e tenta ler da entrada-padrão chamando o procedimento de biblioteca *read*. Esse procedimento envia uma mensagem que contém o descritor de arquivo, o endereço do buffer, e a contagem, para o sistema de arquivos. Essa mensagem é mostrada como (1) na Figura 3-37. Depois de enviar a mensagem, o *shell* bloqueia, esperando a resposta. (Processos de usuário executam somente a primitiva SEND_REC, que combina um SEND com um RECEIVE a partir do processo para o qual foi enviado.)

O sistema de arquivos recebe a mensagem e localiza o nó-i correspondente ao descritor de arquivo especificado. Esse nó-i é para o arquivo de caractere especial */dev/console* e contém os números dos dispositivo primários e secundários para o terminal. O tipo de dispositivo primário para terminais é 4; para o console o número de dispositivo secundário é 0.

O sistema de arquivos pesquisa em seu mapa de dispositivos, *dmap*, para localizar o número da tarefa de terminal. Então, envia uma mensagem para a tarefa de terminal, mostrada como (2) na Figura 3-37. Normalmente, o usuário não terá digitado nada ainda; então, o *driver* de terminal será incapaz de atender à solicitação. Ele envia uma resposta de volta imediatamente para desbloquear o sistema de arquivos e para informar que nenhum caractere está disponível, mostrado como (3) na Figura 3-37. O sistema de arquivos registra o fato de que um processo está esperando entrada de terminal na estrutura do console em *tty_table* e, então, dispara para obter a próxima solicitação para trabalhar. O *shell* do usuário permanece bloquea-

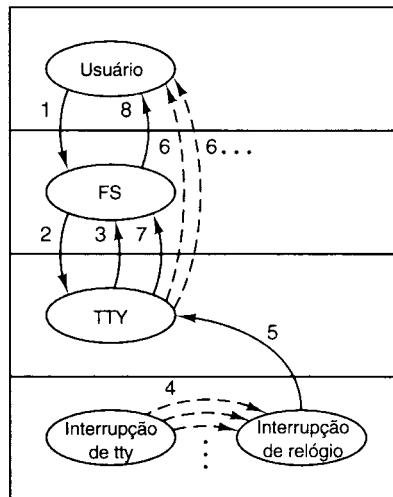



Figura 3-37 Solicitação de leitura de terminal quando nenhum caractere está pendente. FS é o sistema de arquivos. TTY é a tarefa de terminal. O manipulador de interrupções enfileira os caracteres de terminal à medida que eles são inseridos, mas é o manipulador de interrupções de relógio que acorda o TTY.

do até que os caracteres requeridos cheguem naturalmente.

Quando um caractere é finalmente digitado no teclado, isso causa duas interrupções, uma quando a tecla é pressionada e outra quando é liberada. Essa regra também se aplica a teclas modificadoras como CTRL e SHIFT, que não transmitem nenhum dado em si, mas ainda causam duas interrupções por tecla. A interrupção de teclado é IRQ 1, e *_hwint01* no arquivo de código *assembly mpx386.s* ativa *kbd_hw_int* (linha 13123), que, por sua vez, chama *scan_keyboard* (linha 13432) para extrair o código da tecla a partir do hardware do teclado. Se o código é para um caractere comum, ele é colocado na fila de entrada do teclado, *ibuf*, se a interrupção foi gerada por uma tecla sendo pressionada, mas é ignorado se a interrupção foi gerada pela liberação de uma tecla. Os códigos para teclas modificadoras como CTRL e SHIFT são configurados na fila para os dois tipos de interrupção, mas podem ser distinguidas mais tarde por um bit que é ativado somente quando uma tecla é liberada. Note que nesse ponto os códigos recebidos e armazenados em *ibuf* não são códigos em ASCII; são simplesmente os códigos de varredura produzidos pelo teclado IBM. *Kbd_hw_int*, então, ativa um sinalizador, *tty_events* (parte da seção do teclado de *tty_table)*, chama *force_timeout* e retorna.

Diferentemente de algumas outras rotinas de serviço de interrupção, *kbd_hw_int* não envia uma mensagem para acordar para a tarefa de terminal. A chamada para *force_timeout* é indicada pelas linhas tracejadas na figura (4). Essas não são mensagens. Elas configuram a variável *tty_timeout* no espaço de endereço comum para as rotinas de serviço de interrupção. Na próxima interrupção de relógio *clock_handler* descobre que *tty_timeout* indica que é hora para uma chamada a *tty_wakeup* (linha 11452) que, então, envia uma mensagem (5) para a tarefa de terminal. Note que embora o código-fonte para *tty_wakeup* esteja no arquivo *tty_c*, ele executa em resposta à interrupção de relógio, e assim dizemos que a interrupção de relógio envia a mensagem para a tarefa de terminal. Se a entrada estiver chegando rapidamente, diversos códigos de caractere podem ser enfileirados dessa maneira, que é a razão pela qual múltiplas chamadas a *force_timeout* (4) são mostradas na figura.

Ao obter a mensagem de *wakeup*, a tarefa de terminal inspeciona o sinalizador *tty_events* para cada dispositivo do terminal, e, para cada dispositivo que tem o sinalizador configurado, chama *handle_events* (linha 12256). O sinalizador *tty_events* pode sinalizar vários tipos de atividade (embora a mais provável seja uma entrada), de modo que *handle_events* sempre chama as funções específicas de dispositivo tanto para a entrada como para a saída. Para a entrada do teclado, isso resulta em uma chamada a *kb_read* (linha 13165), que monitora códigos de teclado que indicam pressionamento ou liberação das teclas CTRL, SHIFT e ALT e convertem códigos de teclado em códigos ASCII. *Kb_read*, por sua vez, chama *in_process* (linha 12367), que processa os códigos ASCII, levando em conta caracteres especiais e sinalizadores diferentes que podem ser configurados, incluindo se o modo canônico está em efeito. O efeito é normalmente adicionar caracteres à fila de entrada do console em *tty_table*, embora alguns códigos, por exemplo BACKSPACE, tenham outros efeitos. Normalmente, também, *in_process* inicia o ecoamento dos códigos ASCII para o monitor.

Quando o número suficiente de caracteres for recebido, a tarefa de terminal chama o procedimento de linguagem *assembly phys_copy* para copiar os dados para o endereço requerido pelo *shell*. Essa operação também não é uma passagem de mensagem e, por essa razão, é mostrada por linhas tracejadas (6) na Figura 3-37. Mais de uma dessas linhas é mostrada visto que pode haver mais de uma ocorrência dessa operação antes de a solicitação do usuário ser completamente atendida. Quando a operação é, por fim, completada, o *driver* de terminal envia uma mensagem para o sistema de arquivos informando que o trabalho foi feito (7), e o sistema de arquivos reage a essa mensagem enviando outra de volta ao *shell* para desbloqueá-lo (8).

A definição de quando caracteres suficientes deram entrada depende do modo do terminal. No modo canônico, uma solicitação está completa quando uma quebra de linha, um fim de linha ou um código de fim de arquivo é recebido, e, a fim de que o processamento adequado da entrada seja feito, uma linha de entrada não pode exceder o tamanho da fila de entrada. No modo não-canônico, uma leitura pode solicitar um número muito maior de caracteres, e *in_process* pode precisar transferir caracteres mais de uma vez antes que uma mensagem seja retornada para o sistema de arquivos para indicar que a operação está completa.

Note que o *driver* de terminal copia os caracteres reais diretamente de seu próprio espaço de endereço para o do *shell*. Ele não vai primeiro pelo sistema de arquivos. Com E/S de bloco, os dados passam pelo sistema de arquivos, permitindo manter um *cache* dos blocos utilizados mais recentemente. Se acontecer de um bloco requerido estar no *cache*, a solicitação poderá ser satisfeita diretamente pelo sistema de arquivos, sem fazer qualquer E/S de disco

Para E/S de terminal, um *cache* não faz nenhum sentido. Além disso, uma solicitação do sistema de arquivos para um *driver* de disco sempre pode ser satisfeita em, no máximo, algumas centenas de milissegundos, de modo que não há nenhum prejuízo real em fazer o sistema de arquivos simplesmente esperar. A E/S de terminal pode levar horas para completar ou pode nunca se completar (no modo canônico a tarefa de terminal espera uma linha completa, e ela também pode esperar um longo tempo no modo não-canônico, dependendo das configurações de *MIN* e *TIME*). Assim, é inaceitável fazer o sistema de arquivos bloquear até que uma solicitação de entrada do terminal seja satisfeita.

Mais tarde, pode acontecer de o usuário ter digitado rapidamente, e os caracteres estarem disponíveis antes de serem solicitados, a partir de ocorrências anteriores dos eventos 4 e 5. Nesse caso, os eventos 1, 2, 6, 7 e 8 acontecem em rápida sucessão depois da solicitação de leitura; 3 não ocorre de modo algum.

Se acontecer de a tarefa de terminal estar executando no momento de uma interrupção de relógio, nenhuma mensagem poderá ser enviada para ela porque ela não estará esperando. Entretanto, para manter entrada e saída fluindo bem quando a tarefa de terminal está ocupada, os sinalizadores *tty_events* para todos dispositivos terminais são inspecionados em várias outras ocasiões, por exemplo, imediatamente depois de processar e de responder uma mensagem. Assim, os caracteres podem ser adicionados à fila de console sem a ajuda de uma mensagem de *wakeup* do relógio. Se duas ou mais interrupções de relógio ocorrerem antes de o *driver* de terminal terminar o que está fazendo, todos os caracteres são armazenados em *ibuf*, e *tty_flags* é repetidamente ativado. Por fim, a tarefa de terminal recebe uma mensagem; o restante é perdido. Mas como todos os caracteres são armazenados em segurança no buffer, nenhuma entrada digitada é perdida. É até possível que no momento em que uma mensagem for recebida pela tarefa de terminal a entrada esteja completa e uma resposta já tenha sido enviada para o processo de usuário.

O problema do que fazer em um sistema de mensagens não-*bufferizado* (princípio do *rendez-vous*) quando uma rotina de interrupção quer enviar uma mensagem para um processo que está ocupado é inerente a esse tipo de projeto. Para a maioria dos dispositivos, como discos, as interrupções ocorrem somente em resposta a comandos emitidos pelo *driver*; então, somente uma interrupção pode estar pendente em qualquer instante. Os único dispositivos que geram interrupções por si mesmos são o relógio e os terminais (e quando ativada, a rede). O relógio é tratado contando-se os tiques pendentes, de modo que se a tarefa de relógio não receber uma mensagem da interrupção de relógio, ela pode compensar mais tarde. Os terminais são tratados fazendo a rotina de interrupções acumular os caracteres em um buffer e ativar um sinalizador para indicar que caracteres foram recebidos. Se a tarefa de terminal estiver executando, ela verifica todos esses sinalizadores antes de ir dormir e adia ir dormir se houver mais trabalho a fazer.

A tarefa de terminal não é acordada diretamente pela interrupção de terminal devido ao excessivo *overhead* a que isso levaria. O relógio envia uma interrupção à tarefa de terminal no próximo tique após cada interrupção de terminal. A 100 palavras por minuto, um datilógrafo digita menos de 10 caracteres por segundo. Mesmo com um datilógrafo rápido, a tarefa de terminal provavelmente receberá uma mensagem de interrupção para cada caractere digitado no teclado, embora algumas dessas mensagens possam ser perdidas. Se o buffer encher antes de ser esvaziado, caracteres em excesso são descartados, mas a experiência demonstra que, para o teclado, um buffer de 32 caracteres é adequado. No caso de outros dispositivos de entrada, taxas de transmissão de dados mais altas são prováveis — taxas 1.000 ou mais vezes mais rápidas que as de um datilógrafo são possíveis a partir de uma porta serial conectada a um modem de 28.800bps. A essa taxa, aproximadamente 48 caracteres podem ser recebidos pelo modem a cada tique do relógio, mas para permitir compactação de dados na ligação do modem a porta serial conectada ao modem deve ser capaz de tratar pelo menos duas vezes mais. Para linhas seriais, o MINIX oferece um buffer de 1024 caracteres.

Lamentamos que a tarefa de terminal não possa ser implementada sem algum compromisso com os princípios gerais do nosso projeto, mas o método que utilizamos faz o trabalho sem muita complexidade adicional no software e nenhuma perda de desempenho. A alternativa óbvia, jogar fora o princípio do *rendez-vous* e fazer o sistema armazenar todas as mensagens enviadas para destinos que não as estão esperando, é muito complicada e também mais lenta.

Projetistas de sistema reais freqüentemente defrontam-se com o dilema entre utilizar o caso geral, que é elegante o tempo todo, mas utiliza técnicas relativamente lentas, e utilizar técnicas que são normalmente rápidas, mas em um ou dois casos exigem truques para fazê-las funcionar adequadamente. A experiência é realmente a única orientação para determinar qual abordagem é melhor sob certas circunstâncias. Uma considerável experiência em projetos de sistemas operacionais é resumida por Lampson (1984) e Brooks (1975). Embora antigas, tais referências ainda são clássicas.

Completaremos nossa visão geral da entrada de terminal resumindo os eventos que ocorrem quando a tarefa de terminal é ativada pela primeira vez por uma solicitação de leitura e quando é ativada depois de receber entrada do teclado (veja a Figura 3-38). No primeiro caso, quando a mensagem vem à tarefa de terminal solicitando caracteres do teclado, o procedimento principal, *tty_task* (linha 11817), chama *do_read* (linha 11891) para tratar a solicitação. *Do_read* armazena os parâmetros da chamada na entrada do teclado em *tty_table*, no caso de haver um número insuficiente de caracteres armazenado para satisfazer a solicitação.

Ela, então, chama *in_transfer* (linha 12303) para obter qualquer entrada que esteja em espera, e *handle_events* (linha 12256) que, por sua vez, chama *kb_read* (linha 13165) e *in_transfer* mais uma vez, para tentar alimentar o fluxo de entrada com mais alguns caracteres. *Kb_read* chama vários outros procedimentos não mostrados na Figura 3-38 para realizar seu trabalho. O resultado é que qualquer coisa que esteja imediatamente disponível é copiada para o usuário. Se nada estiver disponível, nada é copiado. Se a leitura for completada por *in_transfer* ou por *handle_events*, a mensagem será enviada para o sistema de arquivos quando todos os caracteres forem transferidos; então, o sistema de arquivos pode desbloquear o processo chamador. Se a leitura não se completar (nenhum caractere ou caracteres insuficientes) *do_read* informa de volta o sistema de arquivos, dizendo-lhe que deve suspender o chamador original, ou, se uma leitura não-bloqueadora foi solicitada, cancelar a leitura.

O lado direito da Figura 3-38 resume os eventos que ocorrem quando a tarefa de terminal é acordada subseqüentemente a uma interrupção de teclado. Quando um caractere é digitado, o procedimento de interrupção *kb_hw_int* (linha 13123) coloca o código do caractere recebido no buffer de teclado, ativa um sinalizador para iden-

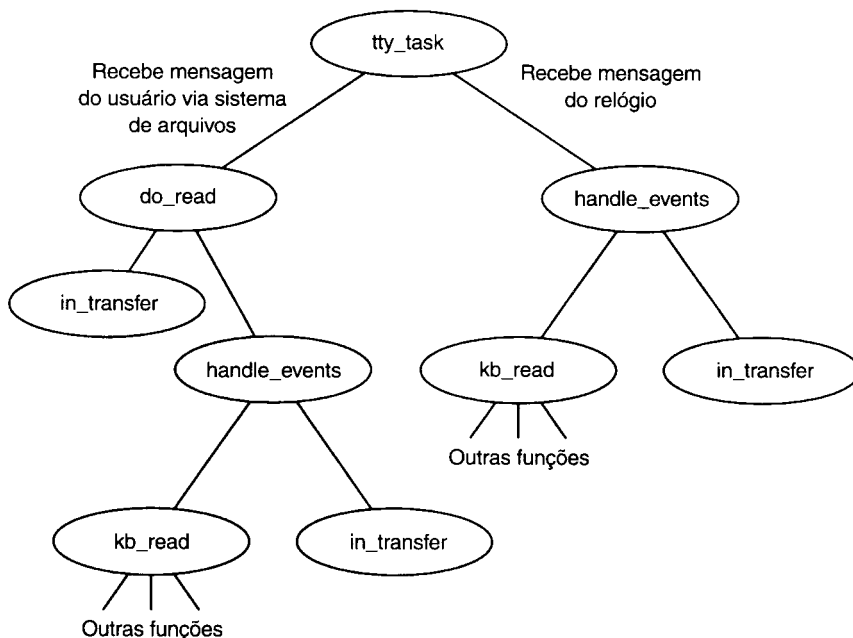



Figura 3-38 Tratamento de entrada no *driver* de terminal. O caminho do ramo esquerdo da árvore é tomado para processar uma solicitação de leitura de caracteres. O do ramo direito é tomado quando uma mensagem "caractere foi digitado" é enviada para o *driver*.

tificar que o dispositivo de console sofreu um evento e, então, determina que um tempo-limite ocorra no próximo tique do relógio. A tarefa de relógio envia uma mensagem à tarefa de terminal informando que algo aconteceu. Ao receber essa mensagem, *tty_task* verifica os sinalizadores de eventos de todos os dispositivos de terminal e chama *handle_event* para cada dispositivo com um sinalizador levantado. No caso do teclado, *handle_event* chama *kb_read* e *in_transfer*, assim como foi feito na recepção da solicitação de leitura original. Os eventos mostrados no lado direito da figura podem ocorrer várias vezes, até que caracteres suficientes sejam recebidos para atender a solicitação aceita por *do_read* depois da primeira mensagem do sistema de arquivos. Se esse tenta iniciar uma solicitação para mais caracteres do mesmo dispositivo antes de a primeira solicitação estar completa, um erro é retornado. Cada dispositivo, naturalmente, é independente; uma solicitação de leitura em nome de um usuário em um terminal remoto é processada separadamente da solicitação para um usuário no console.

As funções não mostradas na Figura 3-38 que são chamadas por *kb_read* incluem *map_key*, que converte os códigos de teclas (códigos de varredura) gerados pelo hardware em códigos ASCII, *make_break*, que monitora o estado das teclas modificadoras como a tecla SHIFT, e *in_process*, que trata complicações como tentativas por parte do usuário de utilizar *backspace* para corrigir um erro, outros caracteres especiais e opções disponíveis em diferentes modos de entrada. *In_process* também chama *echo* (linha 12531) para que os caracteres digitados sejam exibidos na tela.

Saída de Terminal

Em geral, a saída de console é mais simples que a entrada de terminal, pois o sistema operacional está no controle e não precisa preocupar-se com solicitações de saída chegando em momentos inconvenientes. Ademais, como o console do MINIX é um dispositivo mapeado em memória, a saída para o console é particularmente simples. Nenhuma interrupção é necessária: a operação básica é copiar dados de uma região da memória para outra. Por outro lado, todos os detalhes do gerenciamento da exibição, incluindo tratamento de seqüências de escape, devem ser tratadas pelo software do *driver*. Como fizemos com a entrada de teclado na seção anterior, acompanharemos as etapas envolvidas no envio de caracteres para o monitor do console. Vamos supor, neste exemplo, que a escrita é feita no monitor ativo: complicações menores causadas por consoles virtuais serão discutidas mais tarde.

Quando um processo quer imprimir algo, geralmente chama *printf*. *Printf* chama WRITE para enviar uma mensagem para o sistema de arquivos. A mensagem contém um ponteiro para os caracteres a serem impressos (não os próprios caracteres). O sistema de arquivos, então, envia uma mensagem para o *driver* de terminal, que os busca e copia-os para a RAM de vídeo. A Figura 3-39 mostra os procedimentos principais envolvidos na saída.

Quando uma mensagem vem à tarefa de terminal solicitando escrever na tela, *do_write* (linha 11964) é chamada para armazenar os parâmetros na estrutura *tty* do console em *tty_table*. Então, *handle_events* (a mesma função chamada sempre que o sinalizador *tty_events* é encontrado ligado) é chamada. Em cada chamada, essa função chama as rotinas de entrada e saída para o dispositivo selecionado em seu argumento. No caso do monitor do console, isso significa que qualquer entrada de teclado que esteja esperando é processada primeiro. Se houver entrada esperando, os caracteres a serem ecoados são adicionados a quaisquer caracteres que já estão esperando saída. Então, é feita uma chamada a *cons_write* (linha 13729), o procedimento de saída para monitores de memória mapeada. Esse procedimento utiliza *phys_copy* para copiar blocos de caracteres do processo de usuário para um buffer local, possivelmente repetindo esse e os passos seguintes algumas vezes, uma vez que o buffer local armazena somente 64 bytes. Quando o buffer local está cheio, cada byte de 8 bits é transferido para outro buffer, *ramqueue*. Essa é uma matriz de palavras de 16 bits. Bytes alternados são preenchidos com o valor atual do byte de atributo de tela, que determina as cores de primeiro e de segundo planos e outros atributos. Quando possível, os caracteres são transferidos diretamente para *ramqueue*, mas certos caracteres, como os de controle ou aqueles que são partes de seqüências de escape, necessitam de tratamento especial. Tratamento especial também é requerido quando a posição de tela de algum caractere excede a largura da tela ou quando não há mais espaço em *ramqueue*. Nesses casos, *out_char* (linha 13809) é chamada para transferir os caracteres e executar qualquer ação adicional necessária. Por exemplo, *scroll_screen* (linha 13896) é chamada quando uma quebra de linha é recebida enquanto se está endereçando a última linha da tela, e *parse_escape* trata caracteres durante uma seqüência de escape. Normalmente *out_char* chama *flush* (linha 13951) que copia o conteúdo de *ramqueue* para a memória de vídeo, utilizando a rotina de linguagem *assembly mem_vid_copy*. *Flush* também é chamada depois de o último caractere ser transferido para *ramqueue*, a fim de certificar-se de que toda saída armazenada foi exibida. O resultado final de *flush* é comandar o chip da controladora de vídeo 6845 para exibir o cursor na posição correta.

Logicamente, os bytes buscados do processo de usuário poderiam ser gravados na RAM de vídeo um a cada iteração do laço. Entretanto, acumular os caracteres em *ramqueue* e, então, copiar o bloco com uma chamada a *mem_vid_copy* é mais eficiente no ambiente de memória protegida em processadores da classe Pentium. Interessantemente, essa técnica foi introduzida nas primeiras versões do MINIX que executavam em processadores mais antigos sem memória protegida. O precursor de *mem_vid_copy* lidava com um problema de sincronização — com moni-

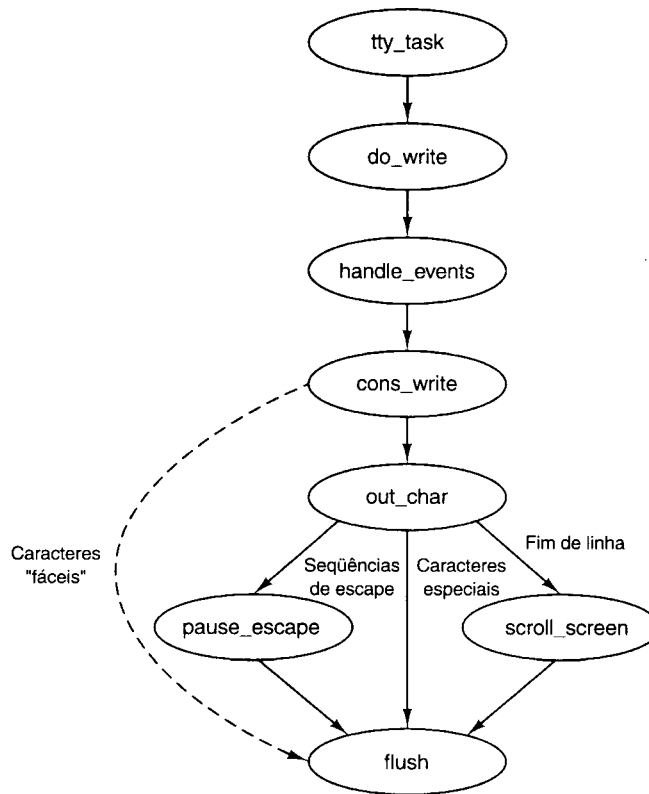



Figura 3-39 Principais procedimentos utilizados na saída de terminal. A linha tracejada indica caracteres copiados diretamente para *ramqueue* por *cons_write*.

tores de vídeo mais antigos, a cópia na memória de vídeo tinha de ser feita quando a tela era limpa durante o retraço vertical do feixe do CRT para evitar gerar lixo visual em toda a tela. O MINIX não oferece mais esse suporte para equipamento obsoleto uma vez que a penalidade no desempenho é muito grande. Entretanto, a versão moderna do MINIX beneficia-se, de outras maneiras, de copiar *ramqueue* como um bloco.

A RAM de vídeo disponível para um console é delimitada na estrutura *console* pelos campos *c_start* e *c_limit*. A posição atual do cursor é armazenada nos campos *c_column* e *c_row*. A coordenada (0, 0) está no canto superior esquerdo da tela, que é onde o hardware começa a preencher a tela. Cada varredura de vídeo começa no endereço dado por *c_org* e continua por 8 × 25 caracteres (4.000 bytes). Em outras palavras, o chip 6845 puxa a palavra no deslocamento *c_org* da RAM de vídeo e exibe o byte do caractere no canto superior esquerdo, utilizando o byte de atributo para controlar a cor, intermitência, etc. Então, ele busca a próxima palavra e exibe o caractere em (1, 0). Esse processo continua até que chegar a (79, 0), momento em que ele inicia a segunda linha na tela, na coordenada (0, 1).

Quando o computador é iniciado pela primeira vez, a tela é limpa, a saída é gravada na RAM de vídeo iniciando na posição *c _start*, e a *c_org* é atribuído o mesmo valor que a *c_start*. Assim, a primeira linha aparece na linha superior da tela. Quando a saída deve ir para uma nova linha, seja porque a primeira linha está cheia seja porque um caractere de nova linha é detectado por *out_char*, a saída é gravada na posição dada por *c_start* mais 80. Por fim, todas as 25 linhas são preenchidas, e a **rolagem** da tela é necessária. Alguns programas, editores, por exemplo, exigem rolar para baixo também, quando o cursor está na linha superior e é necessário mover-se mais para cima no texto.

Há duas maneiras como a rolagem da tela pode ser gerenciada. Na **rolagem por software,** o caractere a ser exibido na posição (0, 0) está sempre na primeira posição na memória do vídeo, a palavra 0 relativamente à posição apontada por *c_start*, e o chip da controladora de vídeo é comandado para exibir essa posição primeiro mantendo o mesmo endereço em *c_org*. Quando a tela for rolada, o conteúdo da posição relativa 80 na RAM de vídeo, o começo da segunda linha na tela, é copiado para a posição relativa 0. A palavra 81 é copiada para a posição relativa 1 e

assim por diante. A seqüência da varredura permanece inalterada, colocando os dados da posição 0 da memória na posição (0, 0) da tela, e a imagem na tela parece ter-se movido para cima uma linha. O custo é que a CPU moveu 80 × 24 = 1920 palavras. Na **rolagem por hardware**, os dados não são movidos na memória; em vez disso, o chip da controladora de vídeo é instruído a iniciar a exibição em um ponto diferente, por exemplo, com os dados na palavra 80. A contabilidade é feita adicionando 80 ao conteúdo de *c_org*, salvando-o para futura referência e gravando esse valor no registrador correto do chip da controladora de vídeo. Isso requer que a controladora seja suficientemente inteligente para referenciar a RAM de vídeo de modo circular, pegando os dados do início da RAM (o endereço em *c_start*) quando alcança o fim (o endereço em *c_limit*), ou que a RAM de vídeo tenha mais capacidade que simplesmente as 80 × 2.000 palavras necessárias para armazenar uma única tela do monitor. Adaptadoras de vídeo mais antigas geralmente têm memória menor, mas são capazes de acessar a RAM de tal modo e fazer a rolagem por hardware. A maioria das novas adaptadoras geralmente tem muito mais memória que necessitaria para exibir uma única tela de texto, mas as controladoras não são capazes de referência circular. Assim, uma adaptadora com 32768 bytes de memória de exibição pode armazenar 204 linhas completas de 160 bytes cada e pode fazer o rolagem por hardware 179 vezes antes de a incapacidade de referência circular tornar-se um problema. Mas, por fim, uma operação de cópia de memória será necessária para mover os dados das últimas 24 linhas de volta à posição na memória do vídeo. Qualquer que seja o método utilizado, uma linha de espaços é copiada para a RAM de vídeo para assegurar que a nova linha no fundo da tela esteja vazia.

Quando consoles virtuais são configurados, a memória disponível dentro de um adaptador de vídeo é dividida igualmente entre o número de consoles desejado por meio da adequada inicialização dos campos *c_start* e *c_limit* para cada console. Isso tem um efeito sobre a rolagem. Em qualquer adaptador suficientemente grande para suportar consoles virtuais, a rolagem por software acontece de vez em quando mesmo que a rolagem por hardware esteja em uso. Quanto menor a quantidade de memória disponível para cada monitor de console, mais freqüentemente a rolagem por software deve ser utilizada. O limite é alcançado quando o máximo número possível de consoles é configurado. Então, cada operação de rolagem é uma operação de rolagem por software.

A posição do cursor em relação ao início da RAM de vídeo pode ser derivada de *c_column* e *c_row*, mas é mais rápido armazená-la explicitamente (em *c_cur*). Quando um caractere é impresso, é colocado na RAM de vídeo na posição *c_cur*, a qual é, então, atualizada, assim como *c_column*. A Figura 3-40 resume os campos da estrutura *console* que afetam a posição atual e a origem da exibição.

Os caracteres que afetam a posição do cursor (p. ex., quebra de linha, *backspace*) são tratados ajustando-se os valores de *column_c*, *c_row* e *c_cur*. Esse trabalho é feito no final de *flush* por uma chamada a *set_6845*, o qual configura os registradores no chip da controladora de vídeo.

O *driver* de terminal suporta seqüências de escape para permitir que editores de tela e outros programas interativos atualizem a tela de uma maneira flexível. As seqüências suportadas são um subconjunto de um padrão ANSI e devem ser adequadas para permitir que muitos programas escritos para outro hardware e para outros sistemas operacionais sejam facilmente portados para MINIX. Há duas categorias de seqüências de escape: aquelas que nunca contêm um parâmetro variável e aquelas que podem conter parâmetros. Na primeira categoria, o único representante suportado pelo MINIX é ESC M, que indexa inversamente a tela, movendo o cursor para cima uma linha e rolando a tela para baixo se o cursor já estiver na primeira linha. A outra categoria pode ter um ou dois parâmetros numéricos. Todas as seqüências nesse grupo começam com ESC [. O "[" é o **introdutor de seqüência de controle.** Uma tabela das seqüências de escape definidas pelo padrão de ANSI e reconhecidas pelo MINIX foi mostrada na Figura 3-36.

O processamento das seqüências de escape não é trivial. Seqüências de escape válidas no MINIX podem ser tão curtas quanto dois caracteres, como em ESC M, ou até 8 caracteres no caso de uma seqüência que aceita dois parâmetros numéricos que podem, cada um, ter um valor de dois dígitos como em ESC [20;60H, que move o cursor para a linha 20, coluna 60. Em uma seqüência que aceita um

Campo	Significado
c_start	Início da memória de vídeo para esse console
c_limit	Limite da memória de vídeo para esse console
c_column	Coluna atual (0-79) com 0 à esquerda
c_row	Fila atual (0-24) com 0 no topo
c_cur	Deslocamento na RAM de vídeo para o cursor
c_org	Posição na RAM apontada pelo registrador de base do 6845

Figura 3-40 Os campos fora a estrutura de console que relaciona à posição atual de tela.

parâmetro, o parâmetro pode ser omitido e, em uma seqüência que aceita dois parâmetros, qualquer um ou ambos podem ser omitidos. Quando um parâmetro é omitido ou quando é utilizado um que está fora do limite válido, ele é substituído pelo padrão. O padrão é o valor válido mais baixo.

Considere as seguintes maneiras como se poderia construir uma seqüência para mover-se para o canto superior esquerdo da tela:

1. ESC [H é aceitável, porque se nenhum parâmetro for dado os parâmetros válidos mais baixos são assumidos.
2. ESC [1;1H enviará corretamente o cursor para a linha 1 e a coluna 1 (com ANSI, os números de linha e de coluna iniciam em 1).
3. Tanto ESC [1; H como ESC [; 1H têm um parâmetro omitido, o que conduz ao padrão 1 como no primeiro exemplo.
4. ESC [0;0H fará o mesmo, uma vez que sendo cada parâmetro inferior ao valor mínimo válido, o mínimo é utilizado no lugar.

Esses exemplos são apresentados não para sugerir que se deve deliberadamente utilizar parâmetros inválidos, mas para mostrar que o código que analisa sintaticamente essas seqüências não é trivial.

O MINIX implementa uma máquina de estados finitos para fazer essa análise sintática. A variável *c_esc_state* na estrutura console normalmente tem um valor de 0. Quando *out_char* detecta um caractere ESC, ela muda *c_esc_state* para 1, e os caracteres subseqüentes são processados por *parse_escape* (linha 13986). Se o próximo caractere for o introdutor de seqüência de controle, o estado 2 é iniciado; caso contrário, a seqüência é considerada completa e *do_escape* (linha 14045) é chamada. No estado 2, contanto que os caracteres que entram sejam numéricos, um parâmetro é calculado multiplicando-se o valor anterior do parâmetro (inicialmente 0) por 10 e adicionando o valor numérico do caractere atual. Os valores dos parâmetros são mantidos em uma matriz e, quando um ponto-e-vírgula é detectado, o processamento muda para a próxima célula na matriz. (A matriz no MINIX tem somente dois elementos, mas o princípio é o mesmo). Quando um caractere não-numérico que não é um ponto-e-vírgula é encontrado, a seqüência é considerada completa e *do_escape* é chamada. O caractere atual na entrada para *do_escape,* então, é utilizado para eleger exatamente que ação tomar e como interpretar os parâmetros, sejam os padrões ou sejam aqueles entrados no fluxo de caracteres. Isso é ilustrado na Figura 3-48.

Mapas de Teclado Carregáveis

O teclado IBM PC não gera códigos ASCII diretamente. As teclas são identificadas individualmente com um número, iniciando com as teclas que estão localizadas no canto esquerdo superior do teclado original do PC —1 para a tecla "ESC", 2 para o "1" e assim por diante. A cada tecla é atribuído um número, incluindo as teclas modificadoras como a tecla SHIFT da esquerda e tecla SHIFT da direita, numeradas como 42 e 54. Quando uma tecla é pressionada, o MINIX recebe o número da tecla como um código de varredura. Um código de varredura também é gerado quando uma tecla é liberada, mas o código gerado na liberação da tecla tem o bit mais significativo ligado (equivalente a adicionar 128 ao número da tecla). Assim, as ações de pressionar e de liberar uma tecla podem ser distinguidas. Monitorando quais teclas modificadoras foram pressionadas e ainda não foram liberadas, um grande número de combinações é possível. Para propósitos normais, naturalmente, combinações de duas teclas, como SHIFT-A ou CTRL-D, são bem gerenciáveis para pessoas que digitam com as duas mãos, mas para ocasiões especiais são possíveis combinações de três teclas (ou mais), por exemplo, CTRL-SHIFT-A ou a bem-conhecida combinação CTRL-ALT-DEL que os usuários de PC reconhecem como a maneira de reinicializar o sistema.

A complexidade do teclado de PC permite muita flexibilidade na maneira como ele é utilizado. Um teclado-padrão tem 47 teclas normais de caractere definidas (26 alfabéticas, 10 numéricas e 11 de pontuação). Se quisermos utilizar três combinações de teclas modificadoras, como CTRL-ALT-SHIFT, podemos suportar um conjunto de caracteres de 376 (8 x 47) elementos. Esse não é de modo algum o limite do que é possível, mas, por enquanto, vamos supor que não queremos distinguir entre as teclas modificadoras da direita e as da esquerda, nem utilizar o teclado numérico nem qualquer tecla de função. De fato, não somos limitados a utilizar somente as teclas CTRL, ALT e SHIFT como modificadoras; poderíamos separar algumas teclas do conjunto de teclas usuais e utilizá-las como modificadoras se quisermos escrever um *driver* que suporte tal sistema.

Os sistemas operacionais que utilizam esses teclados utilizam um **mapa de teclado** para determinar qual código de caractere passar para um programa com base na tecla que está sendo pressionada e as teclas modificadoras em efeito. O mapa de teclado do MINIX logicamente é uma matriz de 128 linhas, representando possíveis valores de código de varredura (esse tamanho foi escolhido para satisfazer aos teclados japoneses; teclados norte-americanos e europeus não têm tantas teclas) e 6 colunas. As colunas representam nenhuma tecla modificadora, a tecla de SHIFT, a tecla Control, a tecla ALT esquerda, a tecla ALT direita e uma combinação de qualquer uma das teclas ALT com a tecla de SHIFT. Há, portanto, 720 ((128 - 6) x 6) códigos de caractere que podem ser gerados por esse esquema, dado um teclado adequado. Isso requer que cada entrada na tabela seja uma quantidade de 16 bits. Para teclados dos EUA as colunas ALT e ALT2 são idênticas. ALT2 é nomeada ALTGR em teclados para outros idiomas, e muitos desses mapas de teclado suportam teclas com três símbolos, utilizando essa tecla como um modificador.

Um mapa de teclado-padrão (determinado pela linha

```
#include keymaps/us-std.src
```

em *keyboard.c*) é compilado no *kernel* do MINIX em tempo de compilação, mas uma chamada

```
ioctl(0, KIOCSMAP, keymap)
```

pode ser utilizada para carregar um mapa diferente no *kernel* a partir do endereço *keymap*. Um mapa de teclado completo ocupa 1536 bytes (128 × 6 × 2). Mapas de teclado extras são armazenados em uma forma compactada. Um programa chamado *genmap* é utilizado para fazer um novo mapa de teclado compactado. Quando compilado, *genmap* inclui o código *keymap.src* para um mapa de teclado particular; então, o mapa é compilado dentro de *genmap*. Normalmente, *genmap* é executado imediatamente depois de ser compilado, momento em que ele dá saída à versão compactada para um arquivo e, então, o executável *genmap* é excluído. O comando *loadkeys* lê um mapa de teclado compactado, expande-o internamente e, então, chama IOCTL para transferir o mapa do teclado para a memória do *kernel*. O MINIX pode executar *loadkeys* automaticamente ao iniciar, e o programa também pode ser invocado pelo usuário em qualquer momento.

O código-fonte para um mapa de teclado define uma grande matriz inicializada e, a fim de economizar espaço, um arquivo de mapa de teclado não é impresso com o código-fonte. A Figura 3-41 mostra na forma de tabela o conteúdo de algumas linhas de *src/kernel/keymaps/usstd.src* para ilustrar vários aspectos dos mapas de teclado. Não há nenhuma tecla no teclado IBM PC que gere um código de varredura 0. A entrada para o código 1, a tecla ESC, mostra que o valor retornado é inalterado quando a tecla de SHIFT ou a tecla CTRL é pressionada, mas que um código diferente é retornado quando uma tecla ALT é pressionada simultaneamente com a tecla ESC. Os valores compilados nas várias colunas são determinados por macros definidas em *include/minix/keymap.h*:

```
#define C(c)      ((c) & 0x1F)       /* Mapeia para o código de control */
#define A(c)      ((c) | 0x80)       /* Liga oito bits (ALT) */
#define CA(c)     A(C(c))            /* CTRL-ALT */
#define L(c)      ((c) | HASCAPS)    /* Adiciona o atributo "Caps Lock ativado" */
```

As primeiras três dessas macros manipulam bits no código do caracter entre aspas para produzir o código necessário a ser retornado para o aplicativo. A última liga o bit HASCAPS no byte alto do valor de 16 bits. Isso é um sinalizador que indica que o estado da variável *capslock* precisa ser verificado, e o código possivelmente modificado antes de retornar. Nas figuras, as entradas para códigos de varredura 2, 13 e 16 mostram como típicas teclas numéricas, de pontuação e alfabéticas são tratadas. Para o código 28, um recurso especial é visto — normalmente a tecla ENTER produz um código CR (0x0D), representado aqui como C('M'). Como o caractere de nova linha em arquivos UNIX é o código LF (0x0A) e, às vezes, é necessário entrar com isso diretamente, esse mapa de teclado oferece uma combinação de CTRL-ENTER, que produz esse código, C('J').

O código de varredura 29 é um dos códigos modificadores e deve ser reconhecido independentemente da outra tecla que é pressionada; então, o valor de CTRL é retornado indiferentemente de qualquer outra tecla que possa ser pressionada. As teclas de função não retornam valores ASCII normais, e a linha para o código de varredura 59 mostra simbolicamente os valores (definidos em *include/minix/keymap.h*) que são retornados para a tecla F1 em combinação com outros modificadores. Esses valores são F1: 0x0110, SF1: 0x1010, AF1: 0x0810, ASF1: 0x0C10 e CF1: 0x0210. A última entrada mostrada na figura, para o códi-

Código de varredura	Caracter	Normal	Shift	ALT1	ALT2	ALT+SHIFT	CTRL
00	nenhum	0	0	0	0	0	0
01	ESC	C('[')	C('[')	CA('[')	CA('[')	CA('[')	C('[')
02	'1'	'1'	'!'	A('1')	A('1')	A('!')	C('A')
13	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

Figura 3-41 Algumas entradas de um arquivo-fonte de mapa de teclado.

go de varredura 127, é típica de muitas entradas próximas do fim da matriz. Para muitos teclados, certamente a maioria dos utilizados na Europa e nas Américas, não há teclas suficientes para gerar todos os possíveis códigos e essas entradas na tabela são preenchidas com zero.

Fontes Carregáveis

Os primeiros PCs tinham os padrões para gerar caracteres em uma tela de vídeo armazenados somente em ROM, mas os monitores utilizados em sistemas modernos oferecem RAM nos adaptadores de vídeo na qual padrões personalizados para o gerador de caracteres podem ser carregados. Isso é suportado pelo MINIX com uma operação IOCTL

```
ioctl (0, TIOCSFON, font)
```

O MINIX suporta um modo de vídeo de 80 colunas x 25 linhas, e os arquivos de fonte contêm 4096 bytes. Cada byte representa uma linha de 8 *pixels* que é iluminada se o valor do bit é 1, e 16 dessas linhas são necessárias para mapear cada caractere. Entretanto, o adaptador de vídeo utiliza 32 bytes para mapear cada caractere, oferecendo resolução mais alta em modos atualmente não-suportados pelo MINIX. O comando *loadfont* é oferecido para converter esses arquivos na estrutura *font* de 8192 bytes referenciada pela chamada IOCTL e utiliza essa chamada para carregar a fonte. Como com os mapas de teclado, uma fonte pode ser carregada em tempo de inicialização ou em qualquer tempo durante operação normal. Entretanto, cada adaptador de vídeo tem uma fonte-padrão embutida em sua ROM que está disponível por padrão. Não há nenhuma necessidade de compilar uma fonte no próprio MINIX, e o único suporte de fonte necessário no *kernel* é o código para executar a operação IOCTL *TIOCSFON*.

3.9.4 Implementação do *Driver* de Terminal Independente de Dispositivo

Nesta seção, começaremos a ver o código-fonte do *driver* de terminal detalhadamente. Vimos quando estudamos os dispositivos de bloco que múltiplas tarefas suportando vários dispositivos diferentes poderiam compartilhar uma base comum de software. O caso com os dispositivos terminais é semelhante, mas com a diferença de que há uma tarefa de terminal que suporta vários tipos diferentes de dispositivo terminal. Aqui começaremos com o código independente de dispositivo. Em seções posteriores veremos o código dependente de dispositivo para o teclado e para o monitor de console mapeado em memória.

Estruturas de Dados da Tarefa de Terminal

O arquivo *tty.h* contém definições utilizadas pelos arquivos de C que implementam os *drivers* de terminal. A maioria das variáveis declaradas nesse arquivo é identificada pelo prefixo *tty_*. Há também uma dessas variáveis declarada em *glo.h* como *EXTERN*. É *tty_timeout*, que é utilizada pelos manipuladores de interrupções do relógio e do terminal

Dentro de *tty.h*, as definições dos sinalizadores *O_NOCTTY* e *O_NONBLOCK* (que são argumentos opcionais para a chamada OPEN) são duplicatas das definições em *include/fcntl.h*, mas repetidas aqui a fim de não precisarmos incluir outro arquivo. Os tipos *devfun_t* e *devfunarg_t* (linhas 11611 e 11612) são utilizados para definir ponteiros para funções, a fim de fornecer chamadas indiretas utilizando um mecanismo semelhante ao que vimos no código para o laço principal dos *drivers* de disco.

A definição mais importante em *tty.h* é a estrutura *tty* (linhas 11614 a 11668). Há uma estrutura dessa para cada dispositivo de terminal (o monitor e o teclado do console contam como um único terminal). A primeira variável na estrutura *tty*, *tty_events*, é o sinalizador que é ligado quando uma interrupção causa uma mudança que requer que a tarefa de terminal atenda o dispositivo. Quando esse sinalizador é levantado, a variável global *tty_timeout* também é tratada para informar ao manipulador de interrupções de relógio para acordar a tarefa de terminal no próximo tique de relógio.

O restante da estrutura *tty* é organizado para agrupar variáveis que lidam com entrada, com saída, com status e com informações sobre operações incompletas. Na seção de entrada, *tty_inhead* e *tty_intail* definem a fila onde caracteres recebidos são armazenados. *Tty_incount* conta o número de caracteres nessa fila e *tty_eotct* conta linhas ou caracteres, como explicado a seguir. Todas as chamadas específicas de dispositivo são feitas indiretamente, com exceção das rotinas que iniciam os terminais, que são chamadas para configurar os ponteiros utilizados para as chamadas indiretas. Os campos *tty_devread* e *tty_icancel* armazenam ponteiros para código específico de dispositivo para executar as operações de leitura e de cancelamento de entrada. *Tty_min* é utilizado em comparações com *tty_eotct*. Quando o último torna-se igual ao anterior, uma operação de leitura está completa. Durante uma entrada canônica, *tty_min* é configurado como 1 e *tty_eotct* conta as linhas entradas. Durante uma entrada não-canônica, *tty_eotct* conta caracteres e *tty_min* é configurado a partir do campo *MIN* da estrutura *termios*. A comparação das duas variáveis informa quando uma linha está pronta ou quando a contagem mínima de caracteres é alcançada, dependendo do modo.

Tty_time armazena o valor do temporizador que determina quando a tarefa de terminal deve ser acordada pelo manipulador de interrupções de relógio e *tty_timenext* é um ponteiro utilizado para associar em uma lista encadeada os campos *tty_time* ativos. A lista é ordenada sempre que um temporizador é configurado; então, o manipulador de interrupções de relógio somente precisa olhar na

primeira entrada. O MINIX pode suportar muitos terminais remotos, dos quais somente alguns podem ter temporizadores ativados em qualquer momento. A lista de temporizadores ativos torna o trabalho do manipulador de relógio mais fácil do que seria se ele precisasse verificar cada entrada em *tty_table*.

Como o enfileiramento da saída é tratado pelo código específico de dispositivo, a seção de saída de *tty* não declara nenhuma variável e consiste inteiramente de ponteiros para funções específicas de dispositivo que escrevem, ecoam, enviam um sinal de pausa e cancelam saída. Na seção de status os sinalizadores *tty_reprint*, *tty_escape* e *tty_inhibited* indicam que o último caractere visto tem um significado especial; por exemplo, quando um caractere CTRL-V (LNEXT) é visto, *tty_escaped* é configurado como 1 indicando que qualquer significado especial do próximo caractere deve ser ignorado.

A próxima parte da estrutura armazena dados sobre as operações *DEV_READ*, *DEV_WRITE* e *DEV_IOCTL* em progresso. Há dois processos envolvidos em cada uma dessas operações. O servidor que gerencia a chamada de sistema (normalmente o sistema de arquivos) é identificado em *tty_incaller* (linha 11644). O servidor chama a tarefa *tty* em nome de outro processo que precisa fazer uma operação de E/S e esse cliente é identificado em *tty_inproc* (linha 11645). Como descrito na Figura 3-37, durante um READ, caracteres são transferidos diretamente da tarefa de terminal para um buffer dentro do espaço de memória do chamador original. *Tty_inproc* e *tty_in_vir* localizam esse buffer. As próximas duas variáveis, *tty_inleft* e *tty_incum* contam os caracteres ainda necessários e aqueles já transferidos. Conjuntos semelhantes de variáveis são necessários para uma chamada de sistema WRITE. Para IOCTL, pode haver uma transferência imediata de dados entre o processo solicitante e a tarefa; assim, um endereço virtual é necessário, mas não há nenhuma necessidade de variáveis para marcar o progresso de uma operação. Uma solicitação IOCTL pode ser adiada, por exemplo, até que a saída atual esteja completa, mas quando é a hora certa a solicitação é executada em uma única operação. Por fim, a estrutura *tty* inclui algumas variáveis que não entram em nenhuma outra categoria, incluindo ponteiros para as funções para tratar as operações *DEV_IOCTL* e *DEV_CLOSE* no nível de dispositivo, uma estrutura *termios* no estilo POSIX e uma estrutura *winsize* que oferece suporte para dispositivos de exibição baseados em janelas. A última parte da estrutura oferece armazenamento para a própria fila de entrada na matriz *tty_inbuf*. Note que essa é uma matriz de *u16_t*, não de caracteres de 8 bits. Embora aplicativos e dispositivos utilizem códigos de 8 bits para caracteres, a linguagem C requer que a função de entrada *getchar* funcione com um tipo maior de dados de modo que possa retornar um valor simbólico *EOF* além de todos os 256 possíveis valores de byte.

A *tty_table*, uma matriz de estruturas *tty*, é declarada utilizando a macro *EXTERN* (linha 11670). Há um elemento para cada terminal ativado pelas definições *NR_CONS*, *NR_RS_LINES* e *NR_PTYS* em *include/minix/config.h*. Para a configuração discutida neste livro, dois consoles são ativados, mas o MINIX pode ser recompilado para adicionar até duas linhas seriais e até 64 pseudoterminais.

Há uma outra definição *EXTERN* em *tty_h*. *Tty_timelist* (linha 11690) é um ponteiro utilizado pelo temporizador para guardar o início da lista encadeada de campos *tty_time*. O arquivo de cabeçalho *tty_h* é incluído em muitos arquivos, e o espaço de armazenamento para tty *_table* e *tty_timelist* é alocado durante a compilação de *table.c*, do mesmo modo como as variáveis *EXTERN* que são definidas no arquivo de cabeçalho *glo.h*.

No fim de *tty.h*, duas macros, *buflen* e *bufend*, são definidas. Essas são utilizadas freqüentemente no código da tarefa de terminal, que faz muitas operações de cópia de dados de e para os buffers.

O Driver de Terminal Independente de Dispositivo

A tarefa de terminal principal e as funções de suporte independentes de dispositivo estão todas em *tty.h*. Como a tarefa suporta muitos dispositivos diferentes, o número de dispositivo secundário deve ser utilizado para distinguir qual está sendo suportado em uma chamada particular, e eles são definidos nas linhas 11760 a 11764. Seguindo-se a isso, há algumas definições de macro. Se um dispositivo não é iniciado, os ponteiros para as funções específicas de dispositivo desses dispositivos conterão zeros colocados aí pelo compilador de C. Isso torna possível definir a macro *tty_active* (linha 11774) que retorna *FALSE* se um ponteiro nulo é localizado. Naturalmente, o código de inicialização para um dispositivo não pode ser acessado indiretamente se parte de seu trabalho é inicializar os ponteiros que tornam o acesso indireto possível. Nas linhas 11777 a 11783, estão definições de macros condicionais para equiparar as chamadas de inicialização para dispositivos RS-232 ou de pseudoterminais a chamadas a uma função nula quando esses dispositivos não são configurados. *Do_pty*, de maneira semelhante poder ser desativado nessa seção. Isso torna possível omitir o código para esses dispositivos inteiramente se ele não for necessário.

Como há tantos parâmetros configuráveis para cada terminal e pode haver alguns terminais em um sistema em rede, a estrutura *termios_defaults* é declarada e é iniciada com valores padrão (todo os quais são definidos em *include/termios.h)* nas linhas 11803 a 11810. Essa estrutura é copiada para entrada *tty_table* de um terminal sempre que é necessário inicializá-lo ou reinicializá-lo. Os padrões para os caracteres especiais foram mostrados na Figura 3-33. A Figura 3-42 mostra os valores-padrão para os vários sinalizadores. Na linha seguinte, a estrutura *winsize_defaults* é declarada de maneira semelhante. Ela é deixada para ser inicializada com zeros pelo compilador C. Essa é a ação-padrão adequada; ela significa "o tamanho da janela é desconhecido, utilize */etc/termcap*."

Campo	Valores-padrão
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

Figura 3-42 Valores-padrão para os sinalizadores de *termios*.

O ponto de entrada para a tarefa de terminal é *tty_task* (linha 11817). Antes de entrar no laço principal, uma chamada é feita a *tty_init* para cada terminal configurado (no laço na linha 11826) e, então, a mensagem de inicialização do MINIX é exibida (linhas 11829 a 11831). Embora o código-fonte mostre uma chamada para *printf* quando esse código é compilado, a macro que converte chamadas para a rotina de biblioteca *printf* em chamadas para *printk* está em efeito. *Printk* utiliza uma rotina chamada *putk* dentro do *driver* de console, de modo que o sistema de arquivos não está envolvido. Essa mensagem vai apenas para o monitor de console primário e não pode ser redirecionada.

O laço principal nas linhas 11833 a 11884 é, a princípio, como o laço principal de qualquer tarefa — ele recebe uma mensagem, executa um switch no tipo de mensagem para chamar a função apropriada e, então, gera uma mensagem de retorno. Entretanto, há algumas complicações. Em primeiro lugar, muito trabalho é feito por rotinas de interrupção de baixo nível, especialmente no tratamento de entrada de terminal. Na seção anterior vimos que caracteres individuais do teclado são recebidos e *bufferizados* sem enviar uma mensagem à tarefa de terminal para cada caractere. Assim, antes de tentar receber uma mensagem, o laço principal sempre varre toda a *tty_table* para inspecionar o sinalizador *tp->tty_events* de cada terminal e chama *handle_events* conforme necessário (linhas 11835 a 11837), para cuidar dos negócios não-finalizados. Somente quando não há nada exigindo atenção imediata é que uma chamada é feita para receber. Se a mensagem recebida é do hardware uma declaração continue provoca o retorno ao início do laço e a verificação de eventos é repetida.

Em segundo lugar, essa tarefa serve a vários dispositivos. Se uma mensagem recebida é de uma interrupção de hardware, o dispositivo ou os dispositivos que necessitam de serviço são identificados verificando-se os sinalizadores *tp->tty_events*. Se a interrupção não é uma interrupção de hardware, o campo *TTY_LINE* na mensagem é utilizado para determinar qual dispositivo deve responder à mensagem. O número de dispositivo secundário é decodificado por uma série de comparações, por meio das quais *tp* é apontado para a entrada correta em *tty_table* (linhas 11845 até 11864). Se o dispositivo é um pseudoterminal, *do_pty* (*pty.c*) é chamada, e o laço principal é reiniciado. Nesse caso, *do-pty* gera a mensagem de resposta. Naturalmente, se pseudoterminais não estão ativados, a chamada a *do_pty* utiliza a macro fictícia definida anteriormente. Pode-se esperar que tentativas para acessar dispositivos inexistentes não ocorreriam, mas é sempre mais fácil adicionar outra verificação que checa se não há nenhum erro em outra parte no sistema. No caso de o dispositivo não existir ou de não estar configurado, uma mensagem de resposta com uma mensagem de erro *ENXIO* é gerada e, novamente, o controle retorna para o topo do laço.

O resto da tarefa assemelha-se ao que vimos no laço principal de outras tarefas, um switch no tipo de mensagem (linhas 11874 a 11883). A função apropriada para o tipo de solicitação, *do_read, do_write* e assim por diante, é chamada. Em cada caso, a função chamada gera a mensagem de resposta, em vez de passar ao laço principal as informações necessárias para criar a mensagem. Uma mensagem de resposta é gerada no fim do laço principal somente se não foi recebido um tipo válido de mensagem, caso em que uma mensagem de erro *ENXIO* é enviada. Como as mensagens de resposta são enviadas de muitos lugares diferentes dentro da tarefa de terminal, uma rotina comum, *tty_reply*, é chamada para tratar os detalhes de construir mensagens de resposta.

Se a mensagem recebida por *tty_task* é um tipo válido de mensagem, não o resultado de uma interrupção, e não vem de um pseudoterminal, o switch no fim do laço principal despachará uma das funções *do_read, do_write, do_ioctl, do-open, do_close* ou *do_cancel*. Os argumentos para cada uma dessas chamadas são *tp*, um ponteiro para uma estrutura *tty*, e o endereço da mensagem. Antes de ver cada um deles, mencionaremos algumas considerações gerais. Como *tty_task* pode servir a múltiplos dispositivos terminais, essas funções devem retornar rapidamente para que o laço principal possa continuar. Entretanto, *do_read, do_write* e *do_ioctl* podem não ser capazes de completar imediatamente todo o trabalho solicitado. Para permitir que o sistema de arquivos sirva a outra chamada, uma resposta imediata é exigida. Se a solicitação não puder ser completada imediatamente o código *SUSPEND* é retornado no campo de status da mensagem de resposta. Isso corresponde à mensagem marcada como (3) na Figura 3-37 e suspende o processo que iniciou a chamada, enquanto desbloqueia o sistema de arquivos. As mensagens correspondentes a (7) e (8) na figura serão enviadas mais tarde quando a operação puder ser completada. Se a solicitação não puder ser satisfeita ou um erro ocorrer, a contagem de bytes transferidos ou o código de erro é retornado

no campo de status da mensagem de retorno para o sistema de arquivos. Nesse caso, uma mensagem será enviada imediatamente do sistema de arquivos de volta para o processo que fez a chamada original, para acordá-lo.

Ler de um terminal é fundamentalmente diferente de ler de um dispositivo de disco. O *driver* de disco envia um comando para o hardware de disco e os dados acabarão sendo retornados, exceto no caso de uma falha mecânica ou elétrica. O computador pode exibir um aviso de entrada na tela, mas não há meio de forçar uma pessoa sentada ao teclado a começar a digitar. A esse respeito, não há garantia nem mesmo de que uma pessoa estará sentada aí. Para fazer o retorno rápido que é exigido, *do_read* (linha 11891) começa armazenando informação que permitirá que a solicitação seja completada mais tarde, quando e se a entrada chegar. Há alguma verificação de erros a ser feita primeiro. É um erro se o dispositivo ainda estiver esperando entrada para atender uma solicitação anterior ou se os parâmetros na mensagem são nulos (linhas 11901 a 11908). Se esses testes são passados, as informações sobre a solicitação são copiadas para os campos adequados nas entradas *tp->tty_table* do dispositivo nas linhas 11911 a 11915. O último passo, configurar *tp->tty_inleft* com o número de caracteres exigido, é importante. Essa variável é utilizada para determinar quando a solicitação de leitura é satisfeita. No modo canônico, *tp->tty_inleft* é decrementada por um para cada caractere retornado, até que um fim de linha seja recebido, ponto em que é repentinamente reduzida a zero. No modo não-canônico, ela é tratada de maneira diferente, mas em qualquer caso ela é redefinida para zero sempre que a chamada é satisfeita, seja por atingir um tempo-limite seja obtendo pelo menos o número mínimo de bytes exigidos. Quando *tp->tty_inleft* atinge zero, uma mensagem de resposta é enviada. Como veremos, mensagens de resposta podem ser geradas em vários lugares. Às vezes, é necessário verificar se um processo de leitura ainda espera uma resposta; um valor não-zero de *tp->tty_inleft* serve como um sinalizador para tal fim.

No modo canônico, um dispositivo terminal espera entrada até que o número de caracteres solicitados na chamada tenha sido recebido ou o fim de uma linha ou o fim do arquivo seja alcançado. O bit *ICANON* na estrutura *termios* é testado na linha 11917 para ver se o modo canônico está em efeito para o terminal. Se não estiver configurado, os valores de *termios MIN* e *TIME* são verificados para determinar que ação tomar.

Na Figura 3-35, vimos como *MIN* e *TIME* interagem para oferecer maneiras diferentes como uma chamada de leitura comporta-se. *TIME* é testado na linha 11918. Um valor de zero corresponde à coluna esquerda na Figura 3-35 e, nesse caso, nenhum teste adicional é necessário neste ponto. Se *TIME* é diferente de zero, então *MIN* é testado. Se é zero, *settimer* é chamada para iniciar o temporizador que terminará a solicitação *DEV_READ* após algum tempo, mesmo que nenhum byte seja recebido. *Tp->tty_min* é aqui configurado como 1, então, a chamada terminará imediatamente se um ou mais bytes forem recebidos antes de atingir o tempo-limite. Nesse ponto, nenhuma verificação quanto a uma possível entrada ainda foi feita; então, mais de um caractere já pode estar esperando para satisfazer a solicitação. Nesse caso, todos os caracteres que estiverem prontos, até o número especificado na chamada READ, serão retornados logo que seja encontrada a entrada. Se *TIME* e *MIN* forem diferentes de zero, o temporizador tem um significado diferente. O temporizador é utilizado como um temporizador entre caracteres nesse caso. Ele é disparado somente depois de o primeiro caractere ser recebido e é reiniciado após cada caractere sucessivo. *Tp->tty_eotct* conta caracteres no modo não-canônico e se ele é zero na linha 11931, nenhum caractere foi recebido ainda e o temporizador entre bytes é inibido. *Lock* e *unlock* são utilizados para proteger essas duas chamadas para *settimer*, a fim de evitar interrupções de relógio quando *settimer* estiver executando.

Em qualquer caso, na linha 11941, *in_transfer* é chamada para transferir quaisquer bytes já na fila de entrada diretamente para o processo de leitura. Em seguida, há uma chamada para *handle_events*, que pode colocar mais dados na fila de entrada e chamar *in_transfer* novamente. Essa duplicação aparente de chamadas requer alguma explicação. Embora a discussão até agora se tenha dado em termos da entrada de teclado, *do_read* está na parte independente de dispositivo do código e também serve a entradas de terminais remotos conectados por linhas seriais. É possível que a entrada anterior tenha preenchido o buffer de entrada RS-232 até o ponto de a entrada ser inibida. A primeira chamada a *in_transfer* não inicia o fluxo novamente, mas a chamada a *handle_events* pode ter esse efeito. O fato de que ela, então, causa uma segunda chamada a *in_transfer* é somente um bônus. A coisa importante é certificar-se de que o terminal remoto tenha permissão para enviar novamente. Qualquer uma dessas chamadas pode resultar na satisfação da solicitação e no envio da mensagem de resposta para o sistema de arquivos. *Tp->tty_inleft* é utilizada como um sinalizador para ver se a resposta foi enviada; se ele é ainda diferente de zero na linha 11944, *do_read* gera e envia a mensagem de resposta. Isso é feito nas linhas 11949 a 11957. Se a solicitação original especificou uma leitura não-bloqueadora, o sistema de arquivos é instruído a passar um código de erro *EAGAIN* de volta ao chamador original. Se a chamada é uma leitura bloqueadora comum, o sistema de arquivos recebe um código *SUSPEND*, desbloqueando-o, mas dizendo-lhe para deixar o chamador original bloqueado. Nesse caso, o campo *tp->tty_inrepcode* do terminal é configurado como *REVIVE*. Quando e se o READ mais tarde for satisfeito, esse código será colocado na mensagem de resposta ao sistema de arquivos para indicar que o chamador original foi colocado para dormir e necessita ser reanimado.

Do_write (linha 11964) é semelhante a *do_read*, mas mais simples, porque há menos opções com que se preocupar no tratamento de uma chamada de sistema WRITE. Verificações semelhantes às feitas por *do_read* são realizadas para ver se uma escrita anterior ainda não está em pro-

gresso e quais parâmetros de mensagem são válidos e, então, os parâmetros da solicitação são copiados para a estrutura *tty*. *Handle_events* é chamada então, e *pt->tty_outleft* é verificado para ver se o trabalho foi feito (linhas 11991 e 11992). Se tiver sido, uma mensagem de resposta já foi enviada por *handle_events*, e não há nada a fazer. Caso contrário, uma mensagem de resposta é gerada com os parâmetros da mensagem, dependendo de a chamada original WRITE ter sido feita ou não no modo não-bloqueador.

A próxima função, *do_ioctl* (linha 12012), apesar de longa, não é difícil de entender. O corpo de *do_ioctl* são duas declarações **switch**. A primeira determina o tamanho do parâmetro apontado pelo ponteiro na mensagem de solicitação (linhas 12033 a 12064). Se o tamanho não for zero, a validade do parâmetro é testada. O conteúdo não pode ser testado aqui, mas o que pode ser testado é se uma estrutura do tamanho necessário, começando no endereço especificado ajusta-se dentro do segmento especificado para ele estar. O restante da função é outro **switch** no tipo da operação IOCTL requerida (linhas 12075 a 12161). Infelizmente, suportar as operações requeridas pelo POSIX com a chamada IOCTL significa que foi necessário inventar nomes para operações IOCTL que sugerem, mas não duplicam, nomes requeridos pelo POSIX. A Figura 3-43 mostra o relacionamento entre os nomes requeridos pelo POSIX e os nomes utilizados pela chamada IOCTL do MINIX. A operação *TCGETS* serve uma chamada *tcgetattr* pelo usuário e simplesmente retorna uma cópia da estrutura *tp->tty_termios* do dispositivo de terminal. Os quatro tipos de requisição seguintes compartilham código. Os tipos de solicitação *TCSETSW*, *TCSETSF* e *TCSETS* correspondem a chamadas de usuário para a função definida pelo POSIX *tcsetattr* e todas têm a ação básica de copiar uma nova estrutura *termios* para uma estrutura *tty* do terminal. A operação de cópia é feita imediatamente para chamadas *TCSETS* e pode ser feita para as chamadas *TCSETSW* e *TCSETSF* se a saída estiver completa, por uma chamada *phys_copy* para obter os dados do usuário, seguida por uma chamada a *setattr*, nas linhas 12098 e 12099. Se *tcsetattr* foi chamada com um modificador solicitando adiamento da ação até a conclusão da saída atual, os parâmetros para a solicitação são colocados na estrutura *tty* do terminal para posterior processamento se o teste de *tp->tty_outleft* na linha 12084 revelar que saída não está completa. *Tcdrain* suspende um programa até que a saída esteja completa e é traduzido em uma chamada IOCTL do tipo *TCDRAIN*. Se a saída já estiver completa, ela não tem mais nada a fazer. Caso contrário, ela também deve deixar as informações na estrutura *tty*.

A função POSIX *tcflush* descarta dados de entrada não-lidos e/ou de saída não-enviados, de acordo com seu argumento; a tradução IOCTL é simples e direta, consistindo em uma chamada à função *tty_icancel* que serve a todos os terminais, e/ou a função específica de dispositivo apontada por *tp->tty_ocancel* (linhas 12102 a 12109). *Tcflow* é traduzido similarmente de maneira simples e direta em uma chamada IOCTL. Para suspender ou reiniciar a saída, ele configura um valor *TRUE* ou *FALSE* em *tp->tty_inhibited* e, então, configura o sinalizador *tp->tty_events*. Para suspender ou reiniciar uma entrada, ele envia o código apropriado *STOP* (normalmente CTRL-S) ou *START* (CTRL-Q) para o terminal remoto, utilizando a rotina específica de dispositivo de eco apontado por *tp->tty_echo* (linhas 12120 a 12125).

A maior parte do restante das operações tratadas por *do_ioctl* é tratada em uma linha de código, chamando uma função apropriada. Nos casos das operações *KIOCSMAP* (car-

Função do POSIX	Operação do POSIX	Tipo do IOCTL	Parâmetro do IOCTL
tcdrain	(nenhum)	TCDRAIN	(nenhum)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(nenhuma)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(nenhuma)	TCSBRK	int=duração

Figura 3-43 As chamadas do POSIX e as operações IOCTL.

regar mapa de teclado) e *TIOCSFON* (carregar fonte), um teste é feito para certificar-se de que o dispositivo é realmente um console, uma vez que essas operações não se aplicam a outros terminais. Se terminais virtuais estiverem em utilização, o mesmo mapa de teclado e de fonte aplica-se a todos os consoles; o hardware não oferece qualquer meio fácil para fazer o contrário. As operações de tamanho de janela copiam uma estrutura *winsize* entre o processo de usuário e a tarefa de terminal. Note, entretanto, o comentário sob o código para a operação *TIOCSWINSZ*. Quando um processo muda seu tamanho de janela, espera-se que o *kernel* envie um sinal *SIGWINCH* para o grupo do processo em algumas versões do UNIX. O sinal não é requerido pelo POSIX-padrão. Mas qualquer pessoa que pense em utilizar essas estruturas deve considerar adicionar código aqui para iniciar esse sinal.

Os últimos dois casos em *do_ioctl* suportam as funções requeridas pelo POSIX *tcgetpgrp* e *tcsetpgrp*. Não há nenhuma ação associada com esses casos e elas sempre retornam um erro. Não há nada errado com isso. Essas funções suportam **controle de *jobs***, a capacidade de suspender e reiniciar um processo a partir do teclado. O controle de *jobs* não é requerido pelo POSIX e não é suportado pelo MINIX. Entretanto, o POSIX requer tais funções, mesmo quando o controle de *jobs* não é suportado, para assegurar a portabilidade dos programas.

Do_open (linha 12171) tem uma ação básica simples a executar — incrementar a variável *tp->tty_openct* do dispositivo para que ele possa ser verificado quanto a estar aberto. Entretanto, há alguns testes a serem feitos primeiro. O POSIX especifica que para terminais normais o primeiro processo que abrir um terminal é o **líder da sessão** e quando um líder de sessão morre, o acesso ao terminal é revogado para outros processos em seu grupo. Os *daemons* devem ser capazes de gravar mensagens de erro, e se sua saída de erro não for redirecionada para um arquivo, ela deve ir para um dispositivo de exibição que não pode ser fechado. Para esse propósito, existe um dispositivo chamado */dev/log* no MINIX. Fisicamente é o mesmo dispositivo que */dev/console*, mas é endereçado por um número de dispositivo secundário separado e tratado diferentemente. Trata-se de um dispositivo somente para gravação e, assim, *do_open* retorna um erro *EACCESS* se uma tentativa for feita para abri-lo para leitura (linha 12183). O outro teste feito por *do_open* é testar o sinalizador *O_NOCTTY*. Se ele não estiver ativo, e o dispositivo não for */dev/log*, o terminal torna-se o terminal controlador para um grupo de processos. Isso é feito colocando o número de processo do processo que realizou a chamada no campo *tp->tty_pgrp* da entrada em *tty_table*. Seguindo-se a isso, a variável *tp->tty_openct* é incrementada, e a mensagem de resposta é enviada.

Um dispositivo terminal pode ser aberto mais de uma vez, e a próxima função, *do_close* (linha 12198), não tem nada a fazer exceto decrementar *tp->tty_openct*. O teste na linha 12204 frustra uma tentativa de fechar o dispositivo no caso de ele ser */dev/log*. Se essa operação for o último fechamento, a entrada é cancelada chamando *tp->tty_icancel*. Rotinas específicas de dispositivo apontadas por *tp->tty_ocancel* e *tp->tty_close* também são chamadas. Então, vários campos na estrutura *tty* para o dispositivo são retornados aos seus valores padrão, e a mensagem de resposta é enviada.

O último manipulador de tipo de mensagem é *do_cancel* (linha 12220). Esse é invocado quando um sinal é recebido para um processo que está bloqueado, tentando ler ou escrever. Há três estados que devem ser verificados:

1. O processo pode estar lendo quando eliminado.
2. O processo pode estar escrevendo quando eliminado.
3. O processo pode estar suspenso por *tcdrain* até que sua saída esteja completa.

Um teste é feito para cada caso, e a rotina geral *tp->tty_icancel*, ou a rotina específica de dispositivo apontada por *tp->tty_ocancel*, é chamada conforme necessário. No último caso, a única ação necessária é desligar o sinalizador *tp->tty_ioreq*, indicando que a operação IOCTL está completa. Por fim, o sinalizador *tp->tty_events* é ativado, e uma mensagem de resposta é enviada.

Código de Suporte ao Driver de Terminal

Agora que vimos as funções de primeiro nível chamadas no laço principal de *tty_task* é hora de vermos o código que as suporta. Iniciaremos com *handle_events* (linha 12256). Como mencionado anteriormente, em cada interação do laço principal da tarefa de terminal, é verificado o sinalizador *tp->tty_events* para cada dispositivo terminal e *handle_events* é chamada se o sinalizador indicar que é requerida atenção para um terminal em particular. *Do_read* e *do_write* também chamam *handle_events*. Essa rotina deve trabalhar rapidamente. Ela redefine o sinalizador *tp->tty_events* e, então, chama as rotinas específicas do dispositivo para leitura e escrita, utilizando os ponteiros para as funções *tp->tty_devread* e *tp->tty_devwrite* (linhas 12279 para 12282). Essas são chamadas incondicionalmente, porque não há nenhum meio de testar se uma leitura ou uma gravação levantou um sinalizador – foi feita uma escolha de projeto aqui, que verificar dois sinalizadores para cada dispositivo seria mais caro do que fazer duas chamadas cada vez que um dispositivo estivesse ativo. Além disso, na maior parte do tempo um caractere recebido de um terminal deve ser ecoado; assim, as duas chamadas serão necessárias. Como observado na discussão sobre tratamento de chamadas *tcsetattr* por *do_ioctl*, o POSIX pode adiar operações de controle em dispositivos até que a saída atual esteja completa; assim o momento imediatamente após a chamada da função específica de dispo-

sitivo *tty_devwrite* é uma boa hora para cuidar de operações *ioctl*. Isso é feito na linha 12285, onde *do_ioctl* é chamada se houver uma solicitação de controle pendente.

Como o sinalizador *tp->tty_events* é levantado por interrupções, e caracteres podem chegar em um fluxo rápido de um dispositivo rápido, há uma chance de que no momento em que as chamadas para as rotinas de leitura e de gravação específicas de dispositivo e *dev_ioctl* estejam completas, outras interrupções tenham levantado o sinalizador novamente. É dada alta prioridade à tarefa de passar adiante uma entrada a partir do buffer onde a rotina de interrupção colocou-a inicialmente. Assim *handle_events* repete as chamadas às rotinas específicas de dispositivo, contanto que o sinalizador *tp->tty_event* seja encontrado levantado ao fim do laço (linha 12286). Quando o fluxo de entrada pára (também poderia ser de saída, mas é mais provável que as entradas façam exigências tão repetidas), *in_transfer* é chamada para transferir caracteres da fila de entrada ao buffer dentro do processo que iniciou a operação de leitura. A própria *in_transfer* envia uma mensagem de resposta se a transferência completar a solicitação, seja transferindo o número máximo de caracteres requerido seja alcançando o fim de uma linha (no modo canônico). Se ela fizer isso, *tp->tty_left* será zero no retorno para *handle_ events*. Aqui um outro teste é feito e uma mensagem de resposta é enviada se o número de caracteres transferidos alcançou o número mínimo requerido. Testar *tp->tty__inleft* previne que uma mensagem duplicada seja enviada.

A seguir, veremos *in_transfer* (linha 12303), que é responsável por mover dados da fila de entrada no espaço de memória da tarefa para o buffer do processo de usuário que solicitou a entrada. Entretanto, uma cópia simples e direta de bloco não é possível. A fila de entrada é um buffer circular, e os caracteres precisam ser verificados para ver se o fim do arquivo não foi alcançado, ou caso o modo canônico esteja em efeito, se a transferência apenas continua através do fim de uma linha. Ademais, a fila de entrada é uma fila de quantidades de 16 bits, mas o buffer do destinatário é uma matriz de caracteres de 8 bits. Assim um buffer local intermediário é utilizado. Os caracteres são verificados um por um à medida que são colocados no buffer local e quando ele se enche ou quando a fila de entrada foi esvaziada, *phys_copy* é chamada para mover o conteúdo do buffer local para o buffer do processo receptor (linhas 12319 a 12345).

Três variáveis na estrutura *tty*, *tp->tty_inleft, tp->tty_eotct e tp->tty_min*, são utilizadas para decidir se *in_transfer* tem qualquer trabalho a fazer, e as primeiras duas controlam seu laço principal. Como mencionado anteriormente, *tp->tty_inleft* é configurado inicialmente como o número de caracteres requeridos por uma chamada READ. Normalmente, ele é decrementado por um, sempre que um caractere é transferido, mas pode ser repentinamente reduzido a zero quando uma condição que sinaliza o fim de uma entrada é alcançada. Sempre que se torna zero, uma mensagem de resposta para o leitor é gerada, assim ela também serve como um sinalizador para indicar se uma mensagem foi enviada. Portanto, no teste na linha 12314, encontrar *tp->tty_inleft* com valor zero é razão suficiente para abortar a execução de *in_transfer* sem enviar uma resposta.

Na próxima parte do teste, *tp->tty_eotct* e *tp->tty_min* são comparados. No modo canônico, essas duas variáveis referem-se a linhas de entrada completas e, no modo não-canônico, elas referem-se a caracteres. *Tp->tty_eotct* é incrementada sempre que uma "quebra de linha" ou um byte é colocado na fila de entrada, e é decrementado por *in_transfer* sempre que uma linha ou byte é removido da fila. Assim, ele conta o número de linhas ou de bytes que foram recebidos pela tarefa de terminal, mas ainda não passados para um leitor. *Tp->tty_min* indica o número mínimo de linhas (no modo canônico) ou de caracteres (no modo não-canônico) que devem ser transferidos para completar uma solicitação de leitura. Seu valor é sempre 1 no modo canônico e pode ser qualquer valor de 0 até *MAX_INPUT* (255 no MINIX) no modo não-canônico. A segunda metade do teste na linha 12314 faz com que *in_transfer* retorne imediatamente no modo canônico se uma linha completa ainda não foi recebida. A transferência não é feita até que uma linha esteja completa de modo que o conteúdo da fila possa ser modificado se, por exemplo, um caractere ERASE ou um KILL for subseqüentemente digitado pelo usuário antes de a tecla ENTER ser pressionada. No modo não-canônico, ocorrerá um retorno imediato se o número mínimo de caracteres ainda não estiver disponível.

Algumas linhas mais adiante, *tp->tty_inleft* e *tp->tty_eotct* são utilizadas para controlar o laço principal de *in_transfer*. No modo canônico, a transferência continua até que não reste mais uma linha completa na fila. No modo não-canônico, *tp->tty_eotct* é uma contagem de caracteres pendentes. *Tp->tty_min* controla se o laço é iniciado, mas não é utilizada para determinar quando parar. Uma vez que o laço começa, todos os caracteres disponíveis ou o número de caracteres solicitados na chamada original será transferido, o que for menor.

Os caracteres são quantidades de 16 bits na fila de entrada. O código real do caractere a ser transferido para o processo de usuário está nos 8 bits inferiores. A Figura 3-44 mostra como os bits altos são utilizados. Três são utilizados para configurar um sinalizador que indica se o caractere foi precedido por LNEXT (CTRL-V) se significa fim de linha ou se representa um de vários códigos que indicam que uma linha esteja completa. Quatro bits são utilizados para uma contagem para mostrar quanto espaço de tela é utilizado quando o caractere é ecoado. O teste na linha 12322 verifica se o bit *IN_EOF* (D na figura) está ligado. Isso é testado na parte superior do laço interno porque um fim de arquivo (CTRL-D) em si não é transferido para o leitor, nem é contado na contagem de caracteres. Como cada caractere é transferido, uma máscara é aplicada para zerar os 8 bits superiores e somente o valor ASCII nos 8 bits inferiores é transferido para o buffer local (linha 12324).

Figura 3-44 Os campos em um código de caractere na fila de entrada.

Há mais uma maneira de sinalizar o fim da entrada, mas a rotina específica de dispositivo de entrada é que deve determinar se um caractere recebido é uma quebra de linha, CTRL-D ou outro caractere desse tipo e marcar cada um desses caracteres. *In_transfer* somente precisa testar quanto a essa marca, o bit *IN_EOT* (*N* na Figura 3-44), na linha 12340. Se isso for detectado, *tp->tty_eotct* é decrementado. No modo não-canônico, cada caractere é contado dessa maneira à medida que é colocado na fila de entrada e cada caractere também é marcado com o bit *IN_EOT* nesse momento, assim *tp->tty_eotct* conta caracteres que ainda não foram removidos da fila. A única diferença na operação do laço principal de *in_transfer* nos dois diferentes modos está localizada na linha 12343. Aqui *tp->tty_inleft* é zerado em resposta ao fato de um caractere marcado como uma quebra de linha ter sido encontrado, mas somente se o modo canônico estiver em efeito. Portanto, quando o controle retorna para o topo do laço, o laço termina adequadamente depois de uma quebra de linha no modo canônico, mas quebras de linha no modo não-canônico são ignoradas.

Quando o laço termina, normalmente há um buffer local parcialmente preenchido a ser transferido (linhas 12347 a 12353). Então, uma mensagem de resposta é enviada se *tp->tty_inleft* alcançar zero. Esse é sempre o caso no modo canônico, mas se o modo não-canônico estiver em efeito, e o número de caracteres transferido é menor que a solicitação completa, a resposta não é enviada. Isso pode ser confuso se você tiver uma memória boa para detalhes a ponto de lembrar-se de que quando vimos chamadas para *in_transfer* (em *do_read* e *handle_events*), o código seguinte à chamada a *in_transfer* envia uma mensagem de resposta se *in_transfer* retornar tendo transferido mais do que a quantidade especificada em *tp->tty_min*, o que certamente será o caso aqui. A razão por que uma resposta não é feita incondicionalmente a partir de *in_transfer* será vista quando discutirmos a próxima função, que chama *in_transfer* sob um conjunto de circunstâncias diferentes.

Essa próxima função é *in_process* (linha 12367), que é chamada a partir do software específico de dispositivo para tratar o processamento comum que deve ser feito em toda entrada. Seus parâmetros são um ponteiro para a estrutura *tty* do dispositivo de origem, um ponteiro para a matriz de caracteres de 8 bits a ser processada e uma contagem. A contagem é retornada para o chamador. *In_process* é uma função longa, mas suas ações não são complicadas. Ela adiciona caracteres de 16 bits à fila de entrada que posteriormente é processada por *in_transfer*.

Há várias categorias de tratamento fornecidas por *in_transfer*:

1. Caracteres normais são adicionados à fila de entrada, estendidos para 16 bits.
2. Os caracteres que afetam o processamento posterior modificam sinalizadores para sinalizar o efeito, mas não são colocados na fila.
3. Os caracteres que controlam o ecoamento têm efeito imediato sem serem colocados na fila.
4. Os caracteres com significado especial têm códigos como o bit *EOT* adicionados ao seu byte superior à medida que são colocados na fila de entrada.

Vejamos primeiro uma situação completamente normal, um caractere comum, como "x" (código 0x78 em ASCII), digitado no meio de uma linha curta, sem nenhuma seqüência de escape em efeito em um terminal que está configurado com as propriedades-padrão do MINIX. Como recebido do dispositivo de entrada, esse caractere ocupa os bits 0 a 7 na Figura 3-44. Na linha 12385, ele teria seu bit mais significativo, o bit 7, redefinido para zero se o bit *ISTRIP* estivesse ligado, mas o padrão no MINIX é não limpar esse bit, permitindo que códigos completos de 8 bits sejam entrados. De qualquer modo isso não afetaria nosso "x". O padrão do MINIX é permitir processamento estendido da entrada; então, o teste do bit *IEXTEN* em *tp->tty_termios.c_lflag* (linha 12388) é aprovado, mas os testes sucessivos falham sob as condições que postulamos: nenhum caractere de escape está em efeito (linha 12391), essa entrada não é o próprio caractere de escape (linha 12397) e essa entrada não é o caractere *REPRINT* (linha 12405).

Os testes nas próximas várias linhas descobrem que o caractere de entrada não é o caractere especial *_POSIX_VDISABLE*, nem um *CR* ou um *NL*. Por fim, um resultado positivo: o modo canônico está em efeito, o que é o padrão normal (linha 12424). Entretanto, nosso caractere não é *ERASE*, tampouco ele é *KILL*, *EOF* (CTRL-D), *NL* ou *OL*;

portanto, na linha 12457 ainda nada terá acontecido para ele. Aqui se descobre que o bit *IXON* foi ligado, por padrão, permitindo a utilização dos caracteres *STOP* (CTRL-S) e *START* (CTRL-Q), mas nos testes que se fazem quanto a estes nenhuma coincidência é encontrada. Na linha 12478, descobre-se que o bit *ISIG*, que permite a utilização dos caracteres *INTR* e *QUIT*, é ligado por padrão, mas novamente nenhuma coincidência é encontrada.

De fato, a primeira coisa interessante que talvez aconteça a um caractere comum ocorre na linha 12491, onde é feito um teste para ver se a fila de entrada já está cheia. Se esse fosse o caso, o caractere seria descartado neste ponto, uma vez que o modo canônico está em efeito, e o usuário não o veria ecoado na tela. (A declaração continue descarta o caractere, uma vez que ela faz o laço externo reiniciar). Entretanto, como postulamos condições completamente normais para esse exemplo, vamos supor que o buffer não está cheio ainda. O próximo teste, para verificar se processamento especial no modo não-canônico é necessário (linha 12497), falha, causando um salto para frente até a linha 12512. Aqui *echo* é chamada para exibir o caractere para o usuário, uma vez que o bit *ECHO* em *tp->tty_termios.c_lflag* está ligado por padrão.

Por fim, nas linhas 12515 a 12519, o caractere é utilizado sendo colocado no fim da fila de entrada. Dessa vez *tp->tty_incount* é incrementado, mas como este é um caractere comum, não é marcado pelo bit *EOT*, e *tp->tty_eotct* não é alterado.

A última linha no laço chama *in_transfer* se o caractere recém-transferido lotar a fila. Entretanto, sob as condições normais que postulamos para esse exemplo, *in_transfer* não faria nada, mesmo se chamada, uma vez que (supondo que a fila foi servida normalmente, e a entrada anterior foi aceita quando a linha de entrada anterior estava completa) *tp->tty_eotct* é zero, *tp->tty_min* é 1 e o teste no início de *in_transfer* (linha 12314) causa um retorno imediato.

Tendo passado por *in_process* com um caractere comum sob condições normais, voltemos agora ao início de *in_process* e vejamos o que acontece em circunstâncias menos normais. Primeiro, veremos o caractere de escape, que permite que um caractere que normalmente tem um efeito especial seja passado para o processo de usuário. Se um caractere de escape está em efeito, o sinalizador *tp->tty_escaped* é ativado e quando isso é detectado (na linha 12391) o sinalizador é desligado imediatamente e o bit *IN_ESC*, bit V na Figura 3-44 é adicionado ao caractere atual. Isso causa processamento especial quando o caractere é ecoado — caracteres de controle precedidos por escape são exibidos como "^"mais o caractere para torná-los visíveis. O bit *IN_ESC* também impede que o caractere seja reconhecido por testes para caracteres especiais. As próximas poucas linhas processam o próprio caractere de escape, o caractere *LNEXT* (CTRL-V por padrão). Quando o código *LNEXT* é detectado, o sinalizador *tp->tty_escaped* é ligado e *rawecho* é chamada duas vezes para dar saída a um "^" seguido de um *backspace*. Isso lembra o usuário ao teclado de que um escape está em efeito e quando o caractere seguinte for ecoado, ele sobrescreverá o "^". O caractere *LNEXT* é um exemplo de caractere que afeta os caracteres seguintes (nesse caso, somente o próximo caractere). Ele não é colocado na fila, e o laço reinicia depois das duas chamadas para *rawecho*. A ordem desses dois testes é importante, tornando possível entrar o próprio caractere *LNEXT* duas vezes em uma linha, para passar a segunda cópia para um processo.

O próximo caractere especial processado por *in_process* é o caractere *REPRINT* (CTRL-R). Quando ele é encontrado, é feita uma chamada a *reprint* (linha 12406), o que faz com que a saída ecoada atualmente seja exibida novamente. O próprio *REPRINT*, então, é descartado sem nenhum efeito sobre a fila de entrada.

Entrar nos detalhes do tratamento de cada caractere especial seria tedioso, e o código-fonte de *in_process* é simples e direto. Mencionaremos somente mais alguns pontos. Um é que a utilização de bits especiais no byte superior do valor de 16 bits colocado na fila de entrada torna fácil identificar uma classe de caracteres que têm efeitos semelhantes. Assim, *EOT* (CTRL-D), *LF* e o caractere alternativo de *EOL* (indefinido por padrão) são marcados pelo bit EOT, o bit D na Figura 3-44 (linhas 12447 a 12453), facilitando posterior reconhecimento. Por fim, justificaremos o comportamento peculiar de *in_transfer* observado anteriormente. Uma resposta não é gerada cada vez que ela termina, apesar de, nas chamadas a *in_transfer* que vimos anteriormente, parecer que uma resposta sempre seria gerada ao retorno. Lembre-se de que a chamada a *in_transfer* feita por *in_process* quando a fila de entrada está cheia (linha 12522) não tem nenhum efeito quando o modo canônico está em efeito. Mas se o processamento não-canônico é desejado, cada caractere é marcado com o bit EOT na linha 12499 e assim cada caractere é contado por *tp->tty_eotct* na linha 12519. Por outro lado, isso causa entrada no laço principal de *in_transfer* quando ela é chamada por causa de uma fila de entrada cheia no modo não-canônico. Em ocasiões assim em que nenhuma mensagem deve ser enviada no término *in_transfer*, é provável que haja mais caracteres lidos depois de retornar para *in_process*. De fato, embora no modo canônico a entrada para um único READ seja limitada pelo tamanho da fila de entrada (255 caracteres no MINIX), no modo não-canônico uma chamada READ é capaz de entregar o número de caracteres *_POSIX_SIZE_MAX* requerido pelo POSIX. Seu valor no MINIX é 32767.

As próximas funções em *tty.c* suportam a entrada de caracteres. *Echo* (linha 12531) trata alguns caracteres de uma maneira especial, mas, em geral, eles são simplesmente exibidos na porção de saída do mesmo dispositivo utilizado para a entrada. A saída de um processo pode estar indo para um dispositivo ao mesmo tempo que a entrada está sendo ecoada, o que torna as coisas confusas se o usuário ao teclado tentar *backspace*. Para lidar com isso, o sinalizador *tp->tty__reprint* sempre é configurado como TRUE pelas rotinas específicas de dispositivo de saída quan-

do é produzida saída normal, de modo que a função chamada para tratar um *backspace* pode informar que a saída misturada foi produzida. Como *echo* também utiliza as rotinas de dispositivo de saída, o valor atual de *tp->tty_reprint* é conservado durante o ecoamento, utilizando a variável local *rp* (linhas 12552 a 12585). Entretanto, se uma nova linha de entrada tiver acabado de começar, *rp* é configurado como *FALSE* em vez de assumir o valor antigo, o que assegura que *tp->tty__reprint* será redefinido quando *echo* terminar.

Você pode ter notado que *echo* retorna um valor, por exemplo, na chamada na linha 12512 em *in_process*

ch = echo (tp, ch)

O valor retornado por *echo* contém o número de espaços utilizados na tela para exibição do eco, que podem ser até oito se o caractere for um *TAB*. Essa contagem é colocada no campo *cccc* na Figura3-44. Caracteres normais ocupam um espaço na tela, mas se um caractere de controle (outro que não *TAB*, *NL* ou *CR*) ou um *DEL* (0x7F) for ecoado, ele será exibido como um "^"mais um caractere ASCII imprimível e ocupará duas posições na tela. Por outro lado *NL* ou *CR* ocupam zero espaços. O eco real é feito por uma rotina específica de dispositivo, naturalmente, e sempre que um caractere deve ser passado para o dispositivo, uma chamada indireta é feita, utilizando *tp->tty_echo*, como, por exemplo, na linha 12580, para caracteres normais.

A próxima função, *rawecho*, é utilizada para pular o tratamento especial feito por *echo*. Ela verifica se o sinalizador *ECHO* está ligado e, se estiver, envia o caractere para a rotina específica de dispositivo *tp->tty_echo* sem qualquer processamento especial. Uma variável local *rp* é utilizada aqui para impedir que a própria chamada de *rawecho* à rotina de saída altere o valor de *tp->tty_reprint*.

Quando um *backspace* é encontrado por *in_process*, a próxima função, *backover* (linha 12607) é chamada. Ela manipula a fila de entrada para remover o elemento anteriormente no início da fila se retroceder for possível — se a fila estiver vazia ou se o último caractere for uma quebra de linha, então retroceder não é possível. Aqui o sinalizador *tp->tty_reprint* mencionado nas discussões sobre *echo* e *rawecho* é testado. Se for *TRUE*, *reprint* será chamada (linha 12618) para por uma cópia limpa da linha de saída na tela. Então, o campo *len* do último caractere exibido (o campo *cccc* da Figura 3-44) é consultado para descobrir quantos caracteres precisam ser excluídos no monitor, e para cada caractere uma seqüência de caracteres *backspace*–espaço–*backspace* é enviada por *rawecho* para remover o caractere indesejável da tela.

Reprint é a próxima função. Além de ser chamada por *backover*, ela pode ser invocada pelo usuário ao pressionar a tecla *REPRINT* (CTRL-R). O laço nas linhas 12651 a 12656 pesquisa para trás pela fila de entrada quanto a última quebra de linha. Se for localizada na última posição preenchida, não há nada a fazer e ela retorna. Caso contrário, ecoa o CTRL-R, que aparece no monitor como a seqüência de dois caracteres "^R" e, então, move-se para a próxima linha e exibe novamente a fila desde a última quebra de linha até o fim.

Agora chegamos em *out_process* (linha 12677). Como *in_process*, ela é chamada por rotinas específicas de dispositivo de saída, mas é mais simples. Ela é chamada pelas rotinas específicas de dispositivo para RS-232 e pseudoterminais, mas não pela rotina de console. *Out_process* trabalha sobre um buffer circular de bytes mas não os remove do buffer. A única alteração feita na matriz é inserir um caractere *CR* à frente de um caractere *NL* no buffer se os bits *OPOST* (ativar processamento de saída) e *ONLCR* (mapear NL para CR-NL) em *tp->tty_termios.oflag* estiverem ligados. Esses dois bits estão ligados por padrão no MINIX. Seu trabalho é manter atualizada a variável *tp->tty_position* na estrutura *tty* do dispositivo. Tabulações e *backspace* complicam a vida.

A próxima rotina é *dev_ioctl* (linha 12763). Suporta *do_ioctl* ao executar a função *tcdrain* e a função *tcsetattr* quando é chamada com a opção *TCSADRAIN* ou com a opção *TCSAFLUSH*. Nesses casos, *do_ioctl* não pode completar a ação imediatamente se a saída estiver incompleta; então, as informações sobre a solicitação são armazenadas nas partes da estrutura *tty* reservadas para operações IOCTL com atraso. Sempre que *handle_events* executa, ela verifica o campo *tp->tty_ioreq* depois de chamar a rotina específica de dispositivo de saída e chama *dev_ioctl* se uma operação estiver pendente. *Dev_ioctl* testa *tp->tty_outleft* para ver se a saída está completa e, se estiver, executa as mesmas ações que *do_ioctl* teria executado imediatamente se não tivesse havido nenhum retardo. Para servir *tcdrain*, a única ação é redefinir o campo *tp->tty_ioreq* e enviar a mensagem de resposta para o sistema de arquivos, dizendo-lhe para acordar o processo que fez a chamada original. A variante *TCSAFLUSH* de *tcsetattr* chama *tty_icancel* para cancelar a entrada. Para ambas as variantes de *tcsetattr*, a estrutura *termios* cujo endereço foi passado na chamada original para IOCTL é copiada para estrutura *tp->tty_termios* do dispositivo. *Setattr* é, então, chamada, sucedida, como com *tcdrain*, pelo envio de uma mensagem de resposta para acordar o chamador original bloqueado.

Setattr (linha 12789) é o próximo procedimento. Como vimos, ela é chamada por *do_ioctl* ou *dev_ioctl* para mudar os atributos de um dispositivo de terminal e por *do_close* para redefinir os atributos de volta às configurações-padrão. *Setattr* é sempre chamada depois de copiar uma nova estrutura *termios* para a estrutura *tty* de um dispositivo, porque meramente copiar os parâmetros não é suficiente. Se o dispositivo está sendo controlado agora no modo não-canônico, a primeira ação é marcar todos os caracteres atualmente na fila de entrada com o bit *IN_EOT*, como teria sido feito quando esses caracteres originalmente foram entrados na fila se o modo não-canônico estivesse em efeito então. É mais fácil simplesmente ir adiante e fazer isso (linhas 12803 a 12809) do que testar se os caracteres já têm o bit configurado. Não há nenhuma maneira de saber quais atributos simplesmente foram mudados e quais ainda mantêm seus valores antigos.

A próxima ação é verificar os valores *MIN* e *TIME*. No modo canônico *tp->tty_min* é sempre 1; isso é configurado na linha 12818. No modo não-canônico, a combinação dos dois valores permite quatro modos de operação diferentes, como vimos na Figura 3-35. Nas linhas 12823 a 12825, *tp->tty_min* é primeiro configurado com o valor passado em *tp->tty_termiso.cc*[*VMIN*], que, então, é modificado se for zero e se *tp->tty_termiso.cc* [VTIME] não for zero.

Por fim, *setattr* certifica-se de que a saída não está parada se o controle XON/XOFF estiver desativado, envia um sinal de *SIGHUP* se a taxa de saída estiver configurada como zero e faz uma chamada indireta à rotina específica de dispositivo apontada por *tp->tty_ioctl* para fazer o que somente pode ser feito no nível de dispositivo.

A próxima função, *tty_reply* (linha 12845) foi mencionada muitas vezes na discussão precedente. Sua ação é absolutamente simples e direta: construir uma mensagem e enviá-la. Se por alguma razão a resposta falhar, um sinal de pânico é emitido. As funções seguintes são igualmente simples. *Sigchar* (linha 12866) solicita que o MM envie um sinal. Se o sinalizador *NOFLSH* não estiver ativo, a fila de entrada será removida — a contagem de linhas ou de caracteres recebidos é zerada, e os ponteiros para o fim e para o início da fila são igualados. Essa é a ação-padrão. Quando um sinal *SIGHUP* está para ser capturado, *NOFLSH* pode ser ligado, para permitir que a entrada e a saída sejam mantidas depois de capturar o sinal. *Tty_icancel* (linha 12891) incondicionalmente descarta entradas pendentes da maneira como descrito para *sigchar* e, além disso, chama a função específica de dispositivo apontada por *tp->tty_icancel,* para cancelar qualquer entrada que possa existir no próprio dispositivo ou que esteja armazenada temporariamente no código de baixo nível.

Tty_init (linha 12905) é chamada uma vez para cada dispositivo quando *tty_task* inicia pela primeira vez. Ela configura padrões. Inicialmente, um ponteiro para *tty_devnop,* uma função fictícia que não faz nada, é associado às variáveis *tp->tty_icancel, tp->tty_ocancel, tp->tty_ioctl* e *tp->tty_close*. *Tty_init,* então, chama uma função de inicialização específica de dispositivo para a categoria apropriada de terminal (console, linha serial ou pseudoterminal). Essas configuram os ponteiros reais para funções específicas de dispositivo chamadas indiretamente. Lembre-se de que se não há absolutamente nenhum dispositivo configurado em uma categoria em particular, uma macro que retorna imediatamente é criada, assim nenhuma parte do código para um dispositivo não-configurado precisa ser compilada. A chamada a *scr_init* inicializa o *driver* de console e também chama a rotina de inicialização para o teclado.

Tty_wakeup (linha 12929), embora curta, é extremamente importante no funcionamento da tarefa de terminal. Sempre que o manipulador de interrupções de relógio executa, isto é, a cada tique do relógio, a variável global *tty_timeout* (definida em *glo.h* na linha 5032) é verificada para ver se contém um valor menor do que o tempo presente. Se tiver, *tty_wakeup* é chamada. *Tty_timeout* é configurado como zero pelas rotinas de serviço de interrupção para *drivers* de terminal, assim *wakeup* é forçada a executar no próximo tique do relógio depois de qualquer interrupção de dispositivo de terminal. *Tty_timeout* também é alterada por *settimer* quando um dispositivo de terminal está servindo uma chamada READ no modo não-canônico e precisa definir um limite de tempo como veremos brevemente. Quando *tty_wakeup* executa, ela primeiro desativa o próximo *wakeup,* atribuindo *TIME_NEVER*, um valor muito distante no futuro, a *tty_timeout*. Então, ela varre a lista encadeada de valores de temporizadores, que é ordenada com os tempos para *wakeups* mais próximos primeiro, até que ela chega a um que é posterior ao tempo atual. Esse é o próximo *wakeup* e ele é colocado em *tty_timeout*. *Tty_wakeup* também configura *tp->tty_min* para esse dispositivo como 0, o que assegura que a próxima leitura será bem-sucedida mesmo que nenhum byte seja recebido, ativa o sinalizador *tp->tty_min* para o dispositivo a fim de assegurar que ela receba atenção quando a tarefa de terminal executar em seguida e remove o dispositivo da lista de temporizadores. Por fim, ela chama *interrupt* para enviar a mensagem de *wakeup* à tarefa. Como mencionado na discussão sobre a tarefa de relógio, *tty_wakeup* é logicamente parte do código de serviço de interrupções de relógio, uma vez que ela é chamada somente a partir daí.

A próxima função, *settimer* (linha 12958) configura temporizadores para determinar quando retornar de uma chamada READ no modo não-canônico. Ela é chamada com os parâmetros *tp,* um ponteiro para uma estrutura *tty*, e *on*, um inteiro que representa *TRUE* ou *FALSE*. Primeiro, a lista encadeada das estruturas *tty* apontadas por *timelist* é varrida, buscando uma entrada existente que coincide com o parâmetro *tp*. Se for encontrado algum, ele é removido da lista (linhas 12968 a 12973). Se *settimer* é chamada para encerrar um temporizador, isso é tudo que ela deve fazer. Se é chamada para definir um temporizador, o elemento *tp->tty_time* na estrutura *tty* do dispositivo é configurado como o tempo atual mais o incremento em décimos de segundo especificado no valor *TIME* da estrutura *termios* do dispositivo. Então, a entrada é colocada na lista, que é mantida ordenada. Por fim, o limite de tempo que acabou de ser colocado na lista é comparado com o valor em *tty_timeout* global, e esta última é substituída se o novo limite de tempo for mais próximo.

Por fim, a última definição em *tty.c* é *tty_devnop* (linha 12992) é uma função de "nenhuma operação" a ser indiretamente endereçada onde um dispositivo não requer um serviço. Vimos *tty_devnop* utilizada em *tty_init* como o valor-padrão atribuído a vários ponteiros de função antes de chamar a rotina de inicialização para um dispositivo.

3.9.5 Implementação do *Driver* de Teclado

Agora nós voltamos para o código dependente de dispositivo que suporta o console do MINIX, que consiste em um

teclado IBM PC e em um dispositivo de exibição mapeado em memória. Os dispositivos físicos que os suportam são inteiramente separados: em um sistema *desktop* padrão, o vídeo utiliza uma placa adaptadora (da qual há pelo menos meia dúzia de tipos básicos) conectada à placa-mãe, enquanto o teclado é suportado por circuitos que fazem parte da placa-mãe, que fazem a interface com um computador de 8 bits de um único chip dentro da unidade de teclado. Os dois subdispositivos requerem suporte de software inteiramente separado, que está localizado nos arquivos *keyboard.c* e *console.c.*

O sistema operacional vê o teclado e o console como partes do mesmo dispositivo */dev/console.* Se houver memória suficiente disponível no adaptador de vídeo, suporte para **consoles virtuais** pode ser compilado, e além de */dev/console* pode haver dispositivos lógicos adicionais, */dev/ttyc1, /dev/ttyc2* e assim por diante. A saída de apenas um console vai para o vídeo em um dado momento, e há somente um teclado a utilizar para entrada em qualquer que seja o console que esteja ativo. Logicamente, o teclado é subserviente ao console, mas isso é manifestado apenas de duas maneiras relativamente menores. Primeiro, *tty_table* contém uma estrutura *tty* para o console, e onde campos separados são oferecidos para entrada e saída, por exemplo, os campos *tty_devread* e *tty_devwrite*, ponteiros para funções em *keyboard.c* e *console.c* são preenchidos em tempo de inicialização. Entretanto, há somente um campo *tty_priv* e ele aponta para as estruturas de dados do console somente. Em segundo lugar, antes de entrar em seu laço principal, *tty_task* chama cada dispositivo lógico uma vez para inicializá-lo. A rotina chamada para */dev/console* está em *console.c*, e o código de inicialização para o teclado é chamado a partir daí. Entretanto, essa hierarquia implícita poderia igualmente ser invertida. Até agora vimos primeiro a entrada antes da saída ao lidar com dispositivos de E/S e continuaremos seguindo esse padrão, discutindo *keyboard.c* nesta seção e deixando a discussão sobre *console.c* para a seção seguinte.

Keyboard.c começa, como a maioria dos arquivos-fonte que vimos, com várias declarações **#include**. Uma dessas, porém, é incomum. O arquivo *keymaps/us-std.src* (incluído na linha 13014) não é um cabeçalho comum; é um arquivo-fonte de C que resulta na compilação de um mapa de teclado-padrão dentro de *keyboard.o* como uma matriz inicializada. O arquivo fonte do mapa de teclado não é incluído nas listagens no fim do livro devido ao seu tamanho, mas algumas entradas representativas são ilustradas na Figura 3-41. Seguindo-se a esse **#include**, vêm macros para definir várias constantes. O primeiro grupo é utilizado em interação de baixo nível com a controladora de teclado. Muitos desses são endereços de portas de E/S ou combinações de bit que têm significado nessas interações. O grupo seguinte inclui nomes simbólicos para teclas especiais. A macro *kb_addr* (linha 13041) sempre retorna um ponteiro para o primeiro elemento da matriz *kb_lines*, uma vez que o hardware IBM suporta somente um teclado. Na próxima linha, o tamanho do buffer de entrada do teclado é simbolicamente definido como *KB_IN_BYTES*, com um valor de 32. As próximas 11 variáveis são utilizadas para armazenar vários estados que devem ser lembrados adequadamente para interpretar o pressionamento de uma tecla. Eles são utilizados de maneiras diferentes. Por exemplo, o valor do sinalizador *capslock* (linha 13046) é alternado entre *TRUE* e *FALSE* cada vez que a tecla CAPS LOCK é pressionada. O sinalizador *shift* (linha 13054) é configurado como *TRUE* quando a tecla *Shift* é pressionada e como *FALSE* quando a tecla *Shift* é liberada. A variável *esc* é configurada quando um código de varredura *escape* é recebido. Ele é sempre zerado na recepção do caractere seguinte.

A estrutura *kb_s* nas linhas 13060 a 13065 é utilizada para monitorar códigos de varredura à medida que eles são entrados. Dentro dessa estrutura, os códigos são armazenados em um buffer circular na matriz *ibuf*, de tamanho *KB_IN_BYTES*. Uma matriz *kb_lines*[*NR_CONS*] dessas estruturas é declarada, uma por console, mas de fato somente o primeiro é utilizado, uma vez que a macro *kb_addr* é sempre utilizada para determinar o endereço do *kb_s* atual. Entretanto, normalmente referenciamos as variáveis dentro de *kb_lines*[0] utilizando um ponteiro para a estrutura, por exemplo, *kb->ihead*, para consistência com a maneira como tratamos os outros dispositivos e para tornar as referências no texto consistentes com aquelas na listagem do código-fonte. Uma quantidade pequena de memória é desperdiçada por causa dos elementos não-utilizados da matriz naturalmente. Entretanto, se alguém fabricar um PC com suporte de hardware para múltiplos teclados, o MINIX está pronto; somente uma modificação da macro *kbaddr* é requerida.

Map_key0 (linha 13084) é definida como uma macro. Ela retorna o código ASCII que corresponde a um código de varredura, ignorando modificadores. Isso é equivalente à primeira coluna (sem *shift*) na matriz do mapa de teclado. Seu grande irmão é *map_key* (linha 13091), que executa o mapeamento completo de um código de varredura para um código ASCII, incluindo contabilização de (múltiplas) teclas modificadoras que são pressionadas ao mesmo tempo que as teclas normais.

A rotina de serviço de interrupção de teclado é *kbd_hw_int* (linha 13123), chamada sempre que uma tecla é pressionada ou é liberada. Ela chama *scan_keyboard* para obter o código de varredura do chip da controladora de teclado. O bit mais significativo do código de varredura é ligado quando a liberação de uma tecla causa a interrupção e, nesse caso, a tecla é ignorada a menos que seja uma das teclas modificadoras. Se a interrupção é causada pelo pressionamento de qualquer tecla ou pela liberação de uma tecla modificadora, o código de varredura bruto é colocado no buffer circular se houver espaço, o sinalizador *tp->tty_events* para o console atual é levantado (linha 13154) e, então, *force_timeout* é chamada para certificar-se de que a tarefa de relógio inicie a tarefa de terminal no próximo tique de relógio. A Figura 3-45 mostra códigos de varredura no buffer para uma curta linha de entrada que contém dois caracteres em caixa alta, cada

42	35	170	18	38	38	24	57	54	17	182	24	19	38	32	28
L+	h	L-	e	l	l	o	SP	R+	w	R-	o	r	l	d	CR

Figure 3-45 Códigos de varredura no buffer de entrada, com as correspondentes teclas pressionadas embaixo, para uma linha de texto inserida no teclado. L+, L-, R+ e R- representam, respectivamente, pressionar e liberar as teclas *Shift* esquerda e direita. O código para uma liberação de tecla é 128 mais o código para o pressionamento da mesma tecla.

um precedido pelo código de varredura para pressionamento de uma tecla *Shift* e seguido pelo código para a liberação da tecla *Shift*.

Quando a interrupção de relógio ocorre, a própria tarefa de terminal executa e, ao encontrar o sinalizador *tp->tty_events* do dispositivo de console ligado, ela chama *kb_read* (linha 13165), a rotina específica de dispositivo, utilizando o ponteiro no campo *tp->tty_devread* da estrutura *tty* do console. *Kb_read* pega códigos de varredura do buffer circular do teclado e coloca códigos ASCII em seu buffer local, que é suficientemente grande para armazenar as seqüências de escape que devem ser geradas em resposta a algum código de varredura do teclado numérico. Ela, então, chama *in_process* no código independente do hardware para colocar os caracteres na fila de entrada. Nas linhas 13181 a 13183, *lock* e *unlock* são utilizados para proteger o decremento de *kb->icount* de uma possível interrupção de teclado, chegando ao mesmo tempo. A chamada para *make_break* retorna o código ASCII como um inteiro. Teclas especiais, como as do teclado numérico e as teclas de função, têm valores maiores que 0xFF neste ponto. Os códigos na faixa de *HOME* a *INSRT* (0x101 a 0x10C, definidas em *include/minix/keymap.h*) resultam de pressionar o teclado numérico e são convertidos em seqüências de escape de 3 caracteres, mostradas na Figura 3-4, utilizando a matriz *numpad_map*. As seqüências, então, são passadas para *in_process* (linhas 13196 a 13201). Códigos mais altos não são passados para *in_process*, mas uma verificação é feita para os códigos de ALT-Seta para a esquerda, ALT-Seta para a direita ou ALT-F1 a ALT-F12, e se um desses é encontrado, *select_console* é chamada para alternar entre consoles virtuais.

Make_break (linha 13222) converte códigos de varredura para ASCII e, então, atualiza as variáveis que monitoram as teclas modificadoras. Primeiro, contudo, ela faz uma verificação quanto à mágica combinação CTRL-ALT-DEL que usuários de PC conhecem bem como a maneira para forçar uma reinicialização sob o MS-DOS. Um desligamento correto é desejável, porém, então antes de tentar iniciar as rotinas da BIOS do PC, um sinal *SIGABRT* é enviado para *init*, o processo-pai de todos os outros processos. *Init* é encarregado de capturar esse sinal e de interpretá-lo como um comando para começar um processo de desligamento correto, antes de causar um retorno para o monitor de inicialização, a partir do qual uma reinicialização completa do sistema ou uma reinicialização do MINIX poder ser comandada. Naturalmente, não é aceitável esperar esse trabalho todas as vezes. A maioria dos usuários entende os perigos de um desligamento brusco e não pressiona CTRL-ALT-DEL até que algo realmente dê errado, e o controle nor-

Tecla	Código de varredura	"ASCII"	Seqüência de escape
Home	71	0x101	ESC [H
Seta para cima	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
–	74	0x10A	ESC [S
Seta para a esquerda	75	0x105	ESC [D
5	76	0x109	ESC [G
Seta para a direita	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
Seta para baixo	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10c	ESC [@

Figura 3-46 Códigos de escape gerados pelo teclado numérico. Enquanto os códigos de varredura para teclas comuns são traduzidos em códigos ASCII, códigos "pseudo-ASCII", com valores maiores do que 0xFF, são atribuídos às teclas especiais.

mal do sistema tornou-se impossível. Nesse ponto, é possível que o sistema esteja tão corrompido que o envio ordenado de um sinal para outro processo pode ser impossível. Essa é a razão por que há uma variável estática *CAD_count* em *make_break*. A maioria das quedas de sistema deixa o sistema de interrupções ainda em funcionamento, de modo que a entrada de teclado ainda pode ser recebida, e a tarefa de relógio pode manter a tarefa de terminal em execução. Aqui o MINIX tira proveito do comportamento esperado dos usuários de computador, que tendem a pressionar teclas repetidamente quando algo não parece funcionar corretamente. Se a tentativa de enviar *SIGABRT* para *init* falhar, e o usuário pressionar CTRL-ALT-DEL duas vezes mais, uma chamada para *wreboot* é feita diretamente, causando um retorno para o monitor sem passar pela chamada a *init*.

A parte principal de *make_break* não é difícil de acompanhar. A variável *make* registra se o código de varredura foi gerado pelo pressionamento ou pela liberação de uma tecla e, então, a chamada para *map_key* retorna o código ASCII para *ch*. Em seguida, vem um switch em *ch* (linhas 13248 13294). Consideraremos dois casos: uma tecla comum e uma tecla especial. Para uma tecla comum, nenhum dos casos coincide, e nada deve acontecer no caso-padrão também (linha 13292), uma vez que os códigos normais de teclas são aceitos somente na fase de pressionamento da tecla. Se, de qualquer maneira, um código de tecla comum é aceito na liberação da tecla, um valor de -1 é colocado no lugar aqui e isso é ignorado pelo chamador, *kb_read*. Uma tecla especial, por exemplo *CTRL*, é identificada no lugar apropriado no switch, neste caso na linha 13249. A variável correspondente, neste caso *control*, registra o estado de *make*, e -1 é substituído pelo código de caractere a ser retornado (e ignorado). O tratamento das teclas *ALT, CALOCK, NLOCK* e *SLOCK* é mais complicado, mas para todas essas teclas especiais o efeito é semelhante: uma variável registra o estado atual (para teclas que são somente efetivas enquanto pressionadas) ou alterna o estado anterior (para as teclas de bloqueio).

Há mais um caso a considerar, o do código *EXTKEY* e a variável *esc*. Isso não deve ser confundido com a tecla ESC no teclado, que retorna o código ASCII 0x1B. Não há como gerar o código *EXTKEY* somente pressionando qualquer tecla ou uma combinação de teclas; ele é o **prefixo de tecla estendida** do teclado do PC, o primeiro byte de um código de varredura de 2 bytes que significa que uma tecla que não era parte do complemento de teclas do PC original mas que tem o mesmo código de varredura, foi pressionada. Em muitos casos, o software trata as duas teclas identicamente. Por exemplo, esse é quase sempre o caso para a tecla "/ " normal e a tecla "/" no teclado de numérico. Em outros casos, pode-se querer distinguir essas teclas. Muitos dos leiautes de teclado, por exemplo, para outros idiomas que não o inglês tratam as teclas ALT esquerda e direita diferentemente, para suportar teclas que devem gerar três códigos diferentes de caractere. As duas teclas ALT geram o mesmo código de varredura (56), mas o código *EXTKEY* precede esse quando o ALT da direita é pressionado. Quando o código *EXTKEY* é retornado, o sinalizador *esc* é ligado. Nesse caso, *make_break* retorna a partir de dentro do switch, pulando assim o último passo antes de um retorno normal, que configura *esc* como zero em qualquer outro caso (linha 13295). Isso tem o efeito de fazer o *esc* efetivo somente para o código mais próximo recebido. Se você conhece as complexidades do teclado de PC como comumente é utilizado, isso lhe será tanto familiar quanto um pouco estranho, porque a BIOS do PC não permite ler o código de varredura para uma tecla ALT e retornar um valor diferente para o código estendido como faz o MINIX.

Set_leds (linha 13303) liga e desliga as luzes que indicam se as teclas *Num Lock, Caps Lock* ou *Scroll Lock* em um teclado de PC estão pressionadas. Um byte de controle, *LED_CODE*, é gravado em uma porta de saída instruindo o teclado que o próximo byte gravado nessa porta é para controle das luzes, e o status das três luzes é codificado em 3 bits desse próximo byte. As próximas duas funções suportam essa operação. *Kb_wait* (linha 13327) é chamada para determinar se o teclado está pronto para obter uma seqüência de comando, e *kb_ack* (linha 13343) é chamada para verificar se o comando foi reconhecido. Esses dois comandos utilizam espera ativa, lendo continuamente até que um código desejado seja visto. Isso não é uma técnica recomendada para tratar a maioria das operações de E/S, mas ligar e desligar luzes do teclado não será feito com muita freqüência e fazer isso de maneira ineficiente não desperdiça muito tempo. Note também que ambos, *kb_wait* e *kb_ack*, poderiam falhar e é possível verificar pelo código de retorno se isso acontecer. Mas ligar a luz no teclado não é suficientemente importante para merecer uma verificação do valor retornado por qualquer chamada, e *set_leds* apenas prossegue cegamente.

Como o teclado é parte do console, sua rotina de inicialização *kb_init* (linha 13359) é chamada a partir de *scr_init* em *console.c*, não diretamente de *tty_init* em *tty.c*. Se consoles virtuais estiverem ativados, (i. e., *NR_CONS* em *include/minix/config.h* é maior que 1), *kb_init* será chamada uma vez para cada console lógico. Depois da primeira vez, a única parte de *kb_init* que é essencial para consoles adicionais está configurando em atribuir o endereço de *kb_read* para *tp->tty_devread* (linha 13367), mas nenhum prejuízo é causado por repetir o resto da função. O resto de *kb_init* inicializa algumas variáveis, configura as luzes no teclado e varre o teclado para certificar-se de que nenhuma tecla pressionada restante seja lida. Quando tudo está pronto, ela chama *put_irq_handler* e *enable_irq*; então, *kbd_hw_int* será executada sempre que uma tecla for pressionada ou liberada.

As próximas três funções são todas bastante simples. *Kbd_loadmap* (linha 13392) é quase trivial. Ela é chamada por *do_ioctl* em *tty.c* para fazer a operação de cópia de um mapa de teclado do espaço do usuário para sobrescrever o mapa de teclado padrão compilado pela inclusão de um arquivo fonte de mapa de teclado no início de *keymap.c*.

Func_key (linha 13405) é chamada a partir de *kb_read* para ver se uma tecla especial destinada a proces-

samento local foi pressionada. A Figura 3-47 resume essas teclas e seus efeitos. O código chamado está localizado em vários arquivos. Os códigos *F1* e *F2* ativam o código em *dmp.c,* que discutiremos na próxima seção. O código *F3* ativa *toggle_scroll,* que está em *console.c,* que também será discutido na próxima seção. As chamadas *CF7, CF8 e CF9* causam chamadas a *sigchar,* em *tty.c.* Quando recursos de rede são adicionados ao MINIX, um **case** adicional, para captar o código *F5*, é adicionado para exibir estatísticas de Ethernet. Há também um grande número de outros códigos de varredura que poderiam ser utilizados para ativar outras mensagens de depuração ou eventos especiais do console.

Scan_keyboard (linha 13432) trabalha no nível de interface de hardware, lendo e gravando bytes em portas de E/S. A controladora de teclado é informada de que um caractere foi lido pela seqüência nas linhas 13440 a 13442, a qual lê um byte, grava-o novamente com o bit mais significativo configurado como 1 e, então, regrava-o com o mesmo bit como 0. Isso impede que os mesmos dados estejam em uma leitura subseqüente. Não há nenhuma verificação de status na leitura do teclado, mas não deve haver nenhum problema de qualquer maneira, uma vez que *scan_keyboard* somente é chamada em resposta a uma interrupção, com a exceção da chamada a partir de *kb_init* para limpar qualquer lixo.

A última função em *keyboard.c* é *wreboot* (linha 13450). Se invocada como um resultado de uma pane de sistema, oferece uma oportunidade para o usuário utilizar as teclas de função para exibir as informações de depuração. O laço nas linhas 13478 a 13487 é outro exemplo de espera ativa. O teclado é lido repetidamente até que um ESC seja digitado. Com certeza, alguém poderia sustentar que uma técnica mais eficiente é necessária depois de uma queda, enquanto se espera um comando para reinicializar. Dentro do laço, *func_key* é chamada para oferecer uma possibilidade de obter as informações que talvez ajudem a analisar a causa de uma queda. Não discutiremos mais detalhes do retorno para o monitor. Os detalhes são muito específicos de hardware e não têm muita relação com o sistema operacional.

3.9.6 Implementação do *Driver* de Vídeo

O vídeo do IBM PC pode ser configurado como vários terminais virtuais, se memória suficiente estiver disponível. Examinaremos o código dependente de dispositivo do console nesta seção. Também veremos as rotinas de *dump* para depuração que utilizam serviços de baixo nível do teclado e do vídeo. Esses oferecem suporte para interação limitada com o usuário no console, quando outras partes normais do sistema MINIX não estão funcionando e podem oferecer informações úteis para monitorar o sistema mesmo próximo de uma falha total.

Suporte específico de hardware para saída de console e vídeo mapeados em memória do PC está em *console.c.* A estrutura *console* é definida nas linhas 13677 a 13693. Em um sentido, essa estrutura é uma extensão da estrutura *tty* definida em *tty.c.* Na inicialização, o campo *tp->tty_priv* da estrutura *tty* de um console recebe um ponteiro para a sua própria estrutura *console.* O primeiro item na estrutura *console* é um ponteiro de volta para a estrutura *tty* correspondente. Os componentes de uma estrutura *console* são o que se esperaria para um monitor de vídeo: variáveis para registrar a linha e a coluna da posição do cursor, os endereços de início e o limite da memória utilizada para exibição, o endereço de memória apontado pelo ponteiro de base do chip da controladora e o endereço atual do cursor. Outras variáveis são utilizadas para gerenciar seqüências de escape. Como os caracteres são inicialmente recebidos como bytes de 8 bits e devem ser combinados com bytes de atributo e transferidos como palavras de 16 bits para memória de vídeo, um bloco a ser transferido é criado em *c_ramqueue*, uma matriz suficientemente grande para armazenar uma linha de 80 colunas inteira de pares atributo-caractere de 16 bits. Cada console virtual precisa de uma estrutura *console.* e o espaço de armazenamento é alocado na matriz *cons_table* (linha 13696). Como fizemos com as estruturas *tty* e *kb_s*, normalmente iremos referenciar os elementos de uma estrutura *console,* utilizando um ponteiro, por exemplo, *cons->c_tty.*

Tecla	Propósito
F1	Exibe a tabela de processos
F2	Exibe os detalhes de utilização de memória pelo processo
F3	Alterna entre rolagem por hardware e por software
F5	Mostra estatísticas de Ethernet (se o suporte de rede for compilado)
CF7	Envia SIGQUIT, mesmo efeito que CTRL-\
CF8	Envia SIGIN_, mesmo efeito que DEL
CF9	Envia SIGKILL, mesmo efeito que CTRL-U

Figura 3-47 As teclas de função detectadas por *func_key( ).*

A função cujo endereço é armazenado em cada entrada *tp->tty_devwrite* do console é *cons_write* (linha 13729). Ela é chamada de apenas um lugar, *handle_events* em *tty.c*. A maioria das outras funções em *console.c* existe para suportar essa função. Quando ela é chamada pela primeira vez depois de um processo de cliente fazer uma chamada WRITE, os dados a sofrerem saída estão no buffer do cliente, o qual pode ser localizado, utilizando os campos *tp->tty_outproc* e *tp->out_vir* na estrutura *tty*. Os campos *tp->tty_outleft* informam quantos caracteres estão para ser transferidos, e o campo *tp->tty_outcum* é inicialmente zero, indicando que nenhum ainda foi transferido. Essa é a situação normal ao entrar em *cons_write*, porque, normalmente, uma vez chamada, ela transfere todos os dados requeridos na chamada original. Entretanto, se o usuário quiser reduzir a velocidade desse processo para revisar os dados na tela, ele pode entrar um caractere *STOP* (CTRL-S) no teclado, o que resulta na ativação do sinalizador *tp->tty_inhibited*. *Cons_write* retorna imediatamente quando esse sinalizador está ativado, mesmo que o WRITE não seja completado. Nesse caso, *handle_events* continuará a chamar *cons_write* e, quando *tp->tty_inhibited* for finalmente redefinido pelo usuário ao digitar um caractere *START* (CTRL-Q), *cons_write* continua com a transferência interrompida.

O único argumento de *cons_write* é um ponteiro para a estrutura *tty* do console particular, assim a primeira coisa que deve ser feita é iniciar *cons*, o ponteiro para a estrutura *console* desse console (linha 13741). Portanto, como *handle_events* chama *cons_write* sempre que executa, a primeira ação é um teste para ver se realmente há trabalho a ser feito. Caso contrário, é feito um retorno rápido (linha 13746). Seguindo-se a isso, o laço principal nas linhas 13751 a 13778 é iniciado. Esse laço é muito semelhante em estrutura ao laço principal de *in_transfer* em *tty.c*. Um buffer local que pode armazenar 64 caracteres é preenchido chamando *phys_copy* para obter os dados do buffer do cliente, o ponteiro para a origem e a contagem são atualizados, e, então, cada caractere no buffer local é transferido para a matriz *cons->c_ramqueue*, junto com um byte de atributo, para posterior transferência à tela por *flush*. Há mais de uma maneira de fazer essa transferência, como vimos na Figura 3-39. *Out_char* poder ser chamada para fazer isso para cada caractere, mas é previsível que nenhum dos serviços especiais de *out_char* seja necessário se o caractere for um caractere visível uma seqüência de escape não está em progresso, a largura da tela não foi excedida e *cons->c_ramqueue* não está cheia. Se o serviço completo de *out_char* não for necessário, o caractere é colocado diretamente em *cons->c_ramqueue*, junto com o byte de atributo (recuperado de *cons->c_attr*), e *cons->c_rwords* (o índice na fila), *cons->c_column* (que armazena a coluna na tela), e *tbuf*, o ponteiro no buffer, são incrementados. Essa colocação direta de caracteres em *cons->c_ramqueue* corresponde à linha tracejada no lado esquerdo da Figura 3-39. Se necessário, *out_char* é chamada (linhas 13766 a 13777). Ela faz toda a contabilidade e adicionalmente chama *flush*, a qual faz a transferência final para memória da tela, quando necessário. A transferência do buffer do usuário para o buffer local e para a fila é repetida, contanto que *tp->tty_outleft* indique que ainda há caracteres a serem transferidos, e o sinalizador *tp->tty_inhibited* não tenha sido ativado. Quando a transferência pára, seja porque a operação WRITE esteja completa, seja porque *tp->tty_inhibited* foi ativado, *flush* é chamada novamente para transferir os últimos caracteres na fila para memória da tela. Se a operação estiver completa (testado vendo se *tp->tty_outleft* é zero), uma mensagem de resposta é enviada, chamando *tty_reply* (linhas 13784 e 13785).

Além das chamadas a *cons_write* a partir de *handle_events*, os caracteres a serem exibidos são também enviados para os consoles por *echo* e *rawecho* na parte independente de hardware da tarefa de terminal. Se o console é o dispositivo atual de saída, chamadas via o ponteiro *tp->tty_echo* são dirigidas para a próxima função, *cons_echo* (linha 13794). *Cons_echo* faz todo seu trabalho, chamando *out_char* e, então, *flush*. À entrada do teclado chega caractere por caractere, e a pessoa que faz a digitação quer ver o eco sem nenhum atraso perceptível, portanto, colocar os caracteres na fila de saída seria insatisfatório.

Agora chegamos a *out_char* (linha 13809). Ela faz um teste para ver se uma seqüência de escape está em progresso, chamando *parse_escape* e, então, retornando imediatamente se estiver (linhas 13814 a 13816). Caso contrário, um switch é iniciado para verificar casos especiais: nulo, *backspace*, caractere de sinal sonoro e assim por diante. O tratamento da maioria desses é fácil de acompanhar. A quebra de linha e a tabulação são os mais complicados, uma vez que envolvem alterações complicadas na posição do cursor na tela e podem exigir rolagem também. O último teste é para o código *ESC*. Se for localizado, o sinalizador *cons->c_esc_state* é ativado (linha 13871) e chamadas futuras a *out_char* são dirigidas para *parse_escape* até que a seqüência esteja completa. No fim, o padrão é assumido para caracteres imprimíveis. Se a largura da tela for excedida, a tela pode necessitar ser rolada, e *flush* é chamada. Antes de um caractere ser colocado na fila de saída um teste é feito para ver se a fila não está cheia, e *flush* é chamada se estiver. Colocar um caractere na fila requer a mesma contabilidade que já vimos em *cons_write*.

A próxima função é *scroll_screen* (linha 13896). *Scroll_screen* manipula tanto a rolagem para cima, a situação normal que deve ser tratada sempre que a linha inferior na tela estiver cheia, e a rolagem para baixo, que ocorre quando comandos de posicionamento do cursor tentarem mover o cursor além da linha superior da tela. Para cada direção de rolagem, há três possíveis métodos. Esses são requeridos para suportar diferentes tipos de placas de vídeo.

Veremos o caso da rolagem para cima. Para começar, o tamanho da tela menos uma linha é atribuído a *chars*. A rolagem por software é realizada por um única chamada a *vid_vid_copy* para mover *chars* caracteres mais para bai-

xo na memória, sendo que o tamanho do movimento é o número de caracteres em uma linha. *Vid_vid_copy* pode fazer referência circular à memória, isto é, se solicitada a mover um bloco de memória que ultrapassa a extremidade superior do bloco atribuído ao monitor de vídeo, ela busca a porção de memória que ultrapassa este limite (*fetch*) a parte de estouro a partir da extremidade inferior do bloco de memória e move-o para um endereço mais alto que a parte que é movida para baixo, tratando o bloco inteiro como uma matriz circular. A simplicidade da chamada esconde uma operação claramente lenta. Mesmo que *vid_vid_copy* seja uma rotina de linguagem *assembly* definida em *klib386.s*, essa chamada requer que a CPU mova 3840 bytes, o que é um trabalho grande mesmo para a linguagem *assembly*.

O método de rolagem por software nunca é o padrão; o operador deve selecioná-lo somente se a rolagem por hardware não funcionar ou não for desejada por alguma razão. A razão disso talvez seja o desejo de utilizar o comando *screendump* para salvar a memória de tela em um arquivo. Quando a rolagem por hardware está em efeito, é provável que *screendump* gere resultados inesperados, porque é provável que o início da memória de tela não coincida com o início da tela visível.

Na linha 13917, a variável *wrap* é testada como a primeira parte de um teste composto. *Wrap* é verdadeira para adaptadoras de vídeo mais antigas que podem suportar rolagem por hardware, e se o teste falhar, a rolagem por hardware simples ocorre na linha 13921, onde o ponteiro da origem utilizado pelo chip da controladora de vídeo, *cons->c_org*, é atualizado para apontar para o primeiro caractere a ser exibido no canto superior esquerdo do vídeo. Se *wrap* for *FALSE*, o teste composto continua com um teste para ver se o bloco a ser movido para cima na operação de rolagem ultrapassa os limites do bloco de memória atribuído para esse console. Se isso acontecer, *vid_vid_copy* é chamada novamente para fazer a movimentação circular do bloco para o início da memória alocada do console, e o ponteiro que indica a origem é atualizado. Se não houver sobreposição, o controle passa para o método simples de rolagem por hardware sempre utilizado por controladoras de vídeo mais antigas. Isso consiste em ajustar *cons->c_org* e, então, colocar a nova origem no registrador correto do chip da controladora. A chamada que realiza isso é feita mais tarde, sendo que é uma chamada para limpar a linha inferior da tela.

O código de rolagem para baixo é muito semelhante ao de rolagem para cima. Por fim, *mem_vid_copy* é chamada para limpar a linha inferior (ou superior) endereçada por *new_line*. Então, *set_6845* é chamada para gravar a nova origem de *cons->c_org* nos registradores apropriados, e *flush* certifica-se de que todas as alterações tornaram-se visíveis na tela.

Mencionamos *flush* (linha 13951) várias vezes. Ela transfere os caracteres na fila para a memória de vídeo utilizando *mem_vid_copy*, atualiza algumas variáveis e, então, assegura-se de que os números de linha e de coluna são razoáveis, ajustando-os se, por exemplo, uma seqüência de escape tentou mover o cursor para uma posição de coluna negativa. Por fim, um cálculo de onde o cursor deveria estar é feito e é comparado com *cons->c_cur*. Se eles não coincidirem e se a memória de vídeo que atualmente está sendo tratada pertence ao console virtual atual, uma chamada a *set_6845* é feita para configurar o valor correto no registrador de cursor da controladora.

A Figura 3-48 mostra como o tratamento de seqüências de escape pode ser representado como uma máquina de estados finitos. Isso é implementado por *parse_escape* (linha 13986) que é chamada no início de *out_char* se *cons->c_esc_state* não for zero. Um ESC em si é detectado por *out_char* e torna *cons->c_esc_state* igual a 1. Quando o próximo caractere é recebido, *parse_escape* prepara para processamento adicional colocando um '\0' em *cons->c_esc_intro*, um ponteiro para o início da matriz de parâmetros, *cons->c_esc_parmv*[0] em *cons->c_esc_parmv* e zeros na própria matriz de parâmetros. Então, o primeiro caractere que se segue a ESC é examinado — valores válidos são "[" ou "M". No primeiro caso, o "[" é copiado para *cons->c_esc_intro*, e o estado é avançado para 2. No segundo caso, *do_escape* é chamada para executar a ação, e o estado de escape é redefinido para zero. Se o primeiro caractere depois do ESC não for válido, ele é ignorado, e os caracteres seguintes são exibidos normalmente.

Quando é encontrada uma seqüência ESC[o próximo caractere entrado é processado pelo código do estado de escape 2. Há três possibilidades nesse ponto. Se o caractere for numérico, seu valor é extraído e adicionado a 10 vezes o valor existente na posição atualmente apontada por *cons->c_esc_parmp*, inicialmente *cons->c_esc_parmv*[*0*] (que foi inicializado com zero). O estado de escape não muda. Isso torna possível entrar uma série de algarismos decimais e acumular um parâmetro numérico grande, embora o valor máximo atualmente reconhecido pelo MINIX seja 80, utilizado pela seqüência que move o cursor para uma posição arbitrária (linhas 14027 a 14029). Se o caractere é um ponto-e-vírgula, o ponteiro para a *string* de parâmetros é avançado, de modo que valores numéricos sucessivos podem ser acumulados no segundo parâmetro (linhas 14031 a 14033). Se fosse necessário modificar *MAX_ESC_PARMS* para alocar uma matriz maior para os parâmetros, esse código não teria de ser alterado para acumular valores numéricos adicionais depois da entrada dos parâmetros adicionais. Por fim, se o caractere não é um algarismo numérico nem um ponto-e-vírgula, *do_escape* é chamada.

Do_escape (linha 14045) é uma das mais longas funções no código-fonte de sistema do MINIX, embora o complemento do MINIX para seqüências de escape reconhecidas seja relativamente modesto. Em toda sua extensão, porém, o código deve ser fácil de acompanhar. Depois de uma chamada inicial, a *flush* para certificar-se de que o monitor de vídeo está inteiramente atualizado, há uma escolha simples, dependendo se o caractere imediatamen-

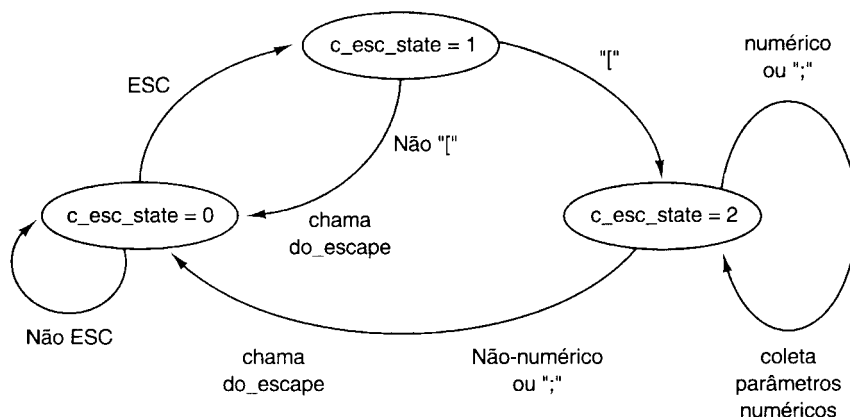



Figura 3-48 Máquina de estados finitos para processamento de seqüências de escape.

te seguinte ao caractere ESC era um introdutor de uma seqüência especial de controle ou não. Se não, há somente uma ação válida, mover o cursor uma linha se a seqüência era ESC M. Note que o teste para o "M" é feito dentro de um **switch** com uma ação-padrão, como uma verificação de validade e em antecipação à adição de outras seqüências que não utilizam o formato ESC [. A ação é típica de muitas seqüências de escape: a variável *cons–>c_row* é inspecionada para determinar se é necessária rolagem. Se o cursor já estiver na linha 0, uma chamada *SCROLL_DOWN* é feita para *scroll_screen*; caso contrário, o cursor é movido para cima uma linha. Este último é realizado simplesmente decrementando *cons->c_row* e, então, chamando *flush*. Se um introdutor de seqüência de controle for localizado, o código seguinte a **else** na linha 14069 é assumido. Um teste é feito quanto a "[", o único introdutor de seqüência de controle atualmente reconhecido pelo MINIX. Se a seqüência for válida, o primeiro parâmetro localizado na seqüência de escape, ou zero se nenhum parâmetro tiver sido entrado, será atribuído a *value* (linha 14072). Se a seqüência for nula, nada acontecerá exceto que o grande **switch** que se segue (linhas 14073 a 14272) é pulado, e o estado de escape é redefinido para zero antes de retornar de *do_escape*. No caso mais interessante em que a seqüência é válida, o **switch** é entrado. Não discutiremos todos os casos: apenas observaremos alguns casos distintos que são bastante representativos dos tipos de ação governados por seqüências de escape.

As primeiras cinco seqüências são geradas, sem nenhum argumento numérico, pelas quatro teclas de "seta" e a tecla *Home* no teclado IBM PC. As duas primeiras, ESC [A e ESC [B, são semelhantes a ESC M, exceto que podem aceitar um parâmetro numérico e mover para cima e para baixo por mais de uma linha e não rolam a tela se o parâmetro especifica um movimento que excede os limites da tela. Nesses casos, *flush* pega solicitações para mover fora dos limites e limita o movimento à última ou à primeira linha, conforme apropriado. As próximas duas seqüências, ESC [C e ESC [D, que movem o cursor para a direita e para a esquerda, são de maneira semelhante limitados por *flush*. Quando geradas pelas teclas de "seta" não há nenhum argumento numérico e, assim, o movimento-padrão de uma linha ou de uma coluna ocorre.

A próxima seqüência, ESC [H, pode ter dois parâmetros numéricos, ESC [20;60H, por exemplo. Os parâmetros especificam uma posição absoluta em vez da posição relativa atual e são convertidos de números baseados em 1 a números baseados em 0 para interpretação adequada. A tecla *Home* gera a seqüência-padrão (nenhum parâmetro) que move o cursor para a posição (1, 1).

As próximas duas seqüências, ESC [*s*J e ESC [*s*K, limpam parte da tela inteira ou da linha atual, dependendo do parâmetro que é dado. Em cada caso, uma contagem de caracteres é calculada. Por exemplo, para ESC [1J, *count* recebe o número de caracteres do início da tela até a posição do cursor; e a contagem de um parâmetro de posição, *dst*, que pode ser o início da tela, *cons->c_org*, ou a posição atual do cursor, *cons >c_cur*, são utilizados como parâmetros para uma chamada a *mem_vid_copy*. Esse procedimento é chamado com um parâmetro que causa o preenchimento da região especificada com a cor de fundo atual.

As próximas quatro seqüências inserem e excluem linhas e espaços na posição do cursor, e suas ações não requerem explicação detalhada. O último caso, ESC [*n*,m (note que o *n* representa um parâmetro numérico, mas o "m" é um caractere literal) tem seu efeito sobre *cons–>c_attr*, o byte de atributo que é intercalado entre os códigos de caractere quando são gravados na memória de vídeo.

A próxima função, *set_6845* (linha 14280), é utilizada sempre que necessário para atualizar o chip da controladora de vídeo. O 6845 tem registradores de 16 bits internos que são programados 8 bits por vez. Gravar um único re-

gistrador requer quatro operações de gravação em porta de E/S. As chamadas *lock* e *unlock* são utilizadas para desativar as interrupções, que podem causar problemas se tiverem permissão para interromper a seqüência. Alguns registradores do chip controlador de vídeo 6845 são mostrados na Figura 3-49.

A função *beep* (linha 14300) é chamada quando um caractere CTRL-G deve sofrer saída. Ela tira proveito do suporte interno fornecido pelo PC para fazer sons enviando uma onda quadrada para o alto-falante. O som é iniciado pelo tipo de manipulação mágica de portas de E/S que apenas programadores de linguagem *assembly* podem gostar, e novamente com alguma preocupação com o fato de que uma parte crítica do processo deve ser protegida contra interrupções. A parte mais interessante do código é a utilização da capacidade da tarefa de relógio de configurar um alarme, que pode ser utilizado para iniciar uma função. A próxima rotina, *stop_beep* (linha 14329), é a rotina cujo endereço é colocado na mensagem para a tarefa de relógio. Ela pára o som depois de o tempo determinado ter passado e também redefine o sinalizador *beeping* que é utilizado para impedir que chamadas supérfluas à rotina *beep* tenham qualquer efeito.

Scr_init (linha 14343) é chamada *NR_CONS* vezes por *tty_init*. A cada vez, seu argumento é um ponteiro para uma estrutura *tty*, um elemento da *tty_table*. Nas linhas 14354 e 14355, *line*, a ser utilizado como índice na matriz *cons_table*, é calculado, é testado quanto à validade, e, se válido, utilizado para iniciar *cons*, o ponteiro para a entrada atual da tabela de console. Nesse ponto, o campo *cons->c_tty* pode ser inicializado com o ponteiro para a estrutura *tty* principal do dispositivo e *tp->tty_priv*, por sua vez, poder ser apontado para a estrutura *console_t* desse dispositivo. Em seguida, *kb_init* é chamada para inicializar o teclado, e, então, os ponteiros para rotinas específicas de dispositivo são configurados, *tp->tty_devwrite* aponta para *cons_write* e *tp->tty_echo* aponta para *cons_echo*. O endereço de E/S do registrador de base da controladora CRT é buscado e o endereço e o tamanho da memória de vídeo são determinados nas linhas 14368 a 14378, e o sinalizador *wrap* (utilizado para determinar como rolar) é configurado de acordo com a classe da controladora de vídeo em uso. Nas linhas 14382 a 14384, o descritor de segmento para a memória de vídeo é inicializado na tabela global de descritores.

Em seguida, vem a inicialização dos consoles virtuais. Cada vez que *scr_init* é chamada, o argumento é um valor diferente de *tp*, e, portanto, um *line* e um *cons* diferentes são utilizados nas linhas 14393 a 14396 para oferecer a cada console virtual sua própria parte da memória de vídeo. Cada tela é, então, limpa, pronta para iniciar e, por fim, o console 0 é selecionado para ser o primeiro ativo.

As rotinas restantes em *console.c* são curtas e simples e iremos revisá-las rapidamente. *Putk* (linha 14408) foi mencionada anteriormente. Ela imprime um caractere em nome de qualquer código vinculado na imagem do *kernel* que precisa do serviço, sem passar pelo sistema de arquivos. *Toggle_scroll* (linha 14429) faz o que seu nome diz; ela comuta o sinalizador que determina se é utilizada rolagem por hardware ou por software. Ela também exibe uma mensagem na posição atual do cursor para identificar o modo selecionado. *Cons_stop* (linha 14442) reinicializa o console para o estado que o monitor de *boot* espera, antes de um desligamento ou de uma reinicialização. *Cons_org0* (linha 14456) é utilizado apenas quando uma mudança no modo de rolagem é forçado pela tecla F3 ou ao preparar o desligamento. *Select_console* (linha 14482) seleciona um console virtual. Ela é chamada com o novo índice e chama *set_6845* duas vezes para fazer a controladora de vídeo exibir a parte adequada da memória de vídeo.

As últimas duas rotinas são altamente específicas de hardware. *Con_loadfont* (linha 14497) carrega uma fonte em um adaptador gráfico, em suporte da operação IOCTL *TIOCSFON*. Ela chama *ga_program* (linha 14540) para fazer uma série de gravações mágicas em uma porta de E/S que faz com que a memória de fonte do adaptador de vídeo, que normalmente não é endereçável pela CPU, seja visível. Então, *phys_copy* é chamada para copiar os dados da fonte para essa área da memória, e outra seqüência mágica é invocada para retornar o adaptador gráfico ao seu modo normal de operação.

Dumps de Depuração

O grupo final de procedimentos que discutiremos na tarefa de terminal foi originalmente projetado somente para utilização temporária em uma depuração do MINIX. Eles podem ser removidos quando esse auxílio não for mais necessário, mas muitos usuários irão achá-los úteis e vão deixá-los onde estão. Eles são particularmente úteis para modificar-se o MINIX.

Como vimos, *func_key* é chamada no início de *kb_read* para detectar códigos de varredura utilizados para controle e depuração. As rotinas de *dump* chamadas quando as te-

Registradores	Função
10 – 11	Tamanho do cursor
12 – 13	Endereço inicial para desenhar a tela
14 – 15	Posição do cursor

Figura 3-49 Alguns registradores do 6845.

clas F1 e F2 são captadas estão em *dmp.c*. A primeira, *p_dmp*, (linha 14613) exibe informações básicas de processo de todos os processos, incluindo algumas informações sobre utilização da memória, quando a tecla F1 é pressionada. A segunda, *map_dmp* (linha 14660), oferece informações mais detalhadas sobre a utilização de memória em resposta a F2. *Proc_name* (linha 14690) suporta *p_dmp*, procurando nomes de processo.

Como esse código está completamente contido dentro do próprio binário do *kernel* e não executa como um processo ou como uma tarefa de usuário, ele com freqüência continua a funcionar corretamente, mesmo depois de uma grande falha do sistema. Naturalmente, essas rotinas são acessíveis somente no console. As informações fornecidas pelas rotinas de *dump* não podem ser redirecionadas para um arquivo nem para qualquer outro dispositivo, assim não são opções a impressão ou a utilização por uma conexão de rede.

Sugerimos que o primeiro passo ao tentar adicionar qualquer melhoria ao MINIX seja estender as rotinas de *dump* para que ofereçam maiores informações sobre o aspecto do sistema que você deseja melhorar.

3.10 A TAREFA DE SISTEMA NO MINIX

Uma conseqüência de fazer os processos de servidor do sistema de arquivos e do gerenciador de memória fora do *kernel* é que ocasionalmente eles têm algumas informações que o *kernel* precisa. Essa estrutura, entretanto, proíbe-os de simplesmente gravá-las em uma tabela do *kernel*. Por exemplo, a chamada de sistema FORK é tratada pelo gerenciador de memória. Quando um novo processo é criado, o *kernel* deve saber sobre ele, para agendá-lo. Como o gerenciador de memória pode informar o *kernel*?

A solução para esse problema é ter uma tarefa de *kernel* que se comunica com o sistema de arquivos e com o gerenciador de memória por meio do mecanismo-padrão de mensagens e que também tenha acesso a todas as tabelas do *kernel*. Essa tarefa, chamada **tarefa de sistema**, está na camada 2 na Figura 2-26 e funciona como as outras tarefas que estudamos nesse capítulo. A única diferença é que ela não controla qualquer dispositivo de E/S. Mas, como as tarefas de E/S, ela implementa uma interface, nesse caso não para o mundo exterior, mas para a parte mais interna do sistema. Ela tem os mesmos direitos concedidos às tarefas de E/S e é compilada com elas na imagem do *kernel* e faz mais sentido estudá-la aqui do que em qualquer outro capítulo.

A tarefa de sistema aceita 19 tipos de mensagens, mostradas na Figura 3-50. O programa principal da tarefa de sistema, *sys_task* (linha 14837), é estruturado como outras tarefas. Ele recebe uma mensagem, despacha para o procedimento de serviço apropriado e, então, envia uma resposta. Agora veremos cada uma dessas mensagens e seus procedimentos de serviço.

A mensagem *SYS_FORK* é utilizada pelo gerenciador de memória para informar o *kernel* de que um novo processo passou a existir. O *kernel* precisa saber disso para agendá-lo. A mensagem contém os números das entradas na tabela de processos correspondentes ao pai e ao filho. O gerenciador de memória e o sistema de arquivos também têm tabelas de processos, com a entrada *k* referenciando o mesmo processo em todas as três. Assim, o gerenciador de memória pode especificar apenas os números de entrada do pai e do filho, e o *kernel* saberá quais processos são indicados.

O procedimento *do_fork* (linha 14877) primeiro faz uma verificação (linha 14886) para ver se o gerenciador de memória está gerando lixo para o *kernel*. O teste utiliza uma macro, *isoksusern*, definida em *proc.h*, para testar se as entradas da tabela de processos para pai e filho são válidas. Testes semelhantes são feitos pela maioria dos procedimentos de serviço em *system.c*. Isso é pura paranóia, mas uma pequena verificação de consistência interna não faz mal nenhum. Então, *do_fork* copia a entrada da tabela de processos do pai para a entrada do filho. Algumas coisas devem ser ajustadas aqui. O filho é liberado de quaisquer sinais pendentes para o pai, e o filho não herda o status de execução do pai. E, naturalmente, todas as informações de contabilidade do filho são configuradas como zero.

Após um FORK, o gerenciador de memória aloca memória para o filho. O *kernel* deve saber onde o filho está localizado na memória para poder configurar os registradores de segmento adequadamente quando executar o filho. A mensagem *SYS_NEWMAP* permite que o gerenciador de memória dê para o *kernel* o mapa de memória de qualquer processo. Essa mensagem também pode ser utilizada depois que uma chamada de sistema BRK alterar o mapa.

A mensagem é tratada por *do_newmap* (linha 14921), que deve primeiro copiar o novo mapa a partir do espaço de endereçamento do gerenciador de memória. O mapa não é contido na mensagem em si porque é muito extenso. Em teoria, o gerenciador de memória poderia informar o *kernel* de que o mapa está no endereço *m*, onde *m* é um endereço ilegal. Não se espera que o gerenciador de memória faça isso, mas o *kernel* verifica de qualquer jeito. O mapa é copiado diretamente para o campo *p_map* da entrada na tabela de processos correspondente que está obtendo o novo mapa. A chamada para *alloc_segments* extrai as informações do mapa e carrega-as nos campos *p_reg* que armazenam os registradores de segmento. Isso não é complicado, mas os detalhes são dependentes do processador e são segregados em uma função a parte por essa razão.

A mensagem *SYS_NEWMAP* é muito utilizada na operação normal de um sistema MINIX. Uma mensagem semelhante, *SYS_GETMAP*, é utilizada somente quando o sistema de arquivos é inicializado. Essa mensagem solicita uma transferência das informações do mapa de processo na direção oposta, do *kernel* para o gerenciador de memória. Ela é executada por *do_getmap* (linha 14957). O có-

Tipo de mensagem	De	Significado
SYS_FORK	MM	Um processo foi criado
SYS_NEWMAP	MM	Instala um mapa de memória para um novo processo
SYS_GETMAP	MM	O gerenciador de memória quer o mapa de memória de um processo
SYS_EXEC	MM	Configura o ponteiro da pilha depois da chamada EXEC
SYS_XIT	MM	Um processo saiu
SYS_GETSP	MM	O gerenciador de memória quer o ponteiro da pilha de um processo
SYS_TIMES	FS	O sistema de arquivos quer os tempos de execução de um processo
SYS_ABORT	Ambos	Pane: o MINIX é incapaz de continuar
SYS_SENDSIG	MM	Envia um sinal para um processo
SYS_SIGRETURN	MM	Limpeza após a conclusão de um sinal
SYS_KILL	FS	Envia sinal para um processo depois da chamada KILL
SYS_ENDSIG	MM	Limpeza depois de um sinal do kernel
SYS_COPY	Ambos	Copia dados entre processos
SYS_VCOPY	Ambos	Copia múltiplos blocos de dados entre processos
SYS_GBOOT	FS	Obter parâmetros de inicialização
SYS_MEM	MM	O gerenciador de memória quer o próximo trecho de memória física
SYS_UMAP	FS	Converte endereço virtual em endereço físico
SYS_TRACE	MM	Executa uma operação da chamada PTRACE

Figura 3-50 Tipos de mensagem aceitos pela tarefa de sistema. (MM = gerenciador de memória; FS = sistema de arquivos)

digo das duas funções é semelhante, diferindo principalmente na troca dos argumentos de origem e de destino utilizados por cada função na chamada a *phys_copy*.

Quando um processo faz uma chamada de sistema EXEC, o gerenciador de memória cria uma nova pilha, contendo os argumentos e o ambiente. Ele passa o ponteiro da pilha resultante para o *kernel* utilizando *SYS_EXEC*, a qual é tratada por *do_exec* (linha 14990). Depois da verificação normal para um processo válido, há um teste do campo *PROC2* na mensagem. Esse campo é utilizado aqui como sinalizador para indicar se os passos da execução do processo estão sendo depurados e não tem nada a ver com identificar um processo. Se a depuração está em efeito, *cause_sig* é chamada para enviar um sinal *SIGTRAP* para o processo. Isso não tem as conseqüências normais desse sinal, que normalmente seriam terminar um processo e causar um *dump* de núcleo. No gerenciador de memória todos os sinais para um processo que está sendo depurado, exceto *SIGKILL*, são interceptados e fazem o processo sinalizado parar de modo que um programa de depuração, então, possa controlar sua execução posterior.

A chamada de EXEC causa uma ligeira anomalia. O processo que invoca a chamada envia uma mensagem para o gerenciador de memória e bloqueia. Com outras chamadas de sistema, a resposta resultante desbloqueia-os. Com EXEC, não há nenhuma resposta, porque a imagem recentemente carregada do núcleo não está esperando uma resposta. Portanto, *do_exec* desbloqueia o próprio processo na linha 15009. A próxima linha torna a nova imagem pronta para executar, utilizando a função *lock_ready* que protege contra uma possível condição de corrida. Por fim, a *string* de comando é salva para que o processo possa ser identificado quando o usuário pressionar a tecla de função F1 para exibir o status de todos os processos.

Os processos podem sair no MINIX tanto fazendo uma chamada de sistema EXIT, que envia uma mensagem para o gerenciador de memória, quanto sendo eliminandos por um sinal. Em ambos os casos, o gerenciador de memória informa o *kernel* por meio da mensagem *SYS_XIT*. O trabalho é feito por *do_xit* (linha 15027), que é mais complicada do que talvez se espere. Cuidar das informações de contabilidade é simples e direto. O temporizador de alarme, se houver um, é eliminado, armazenando um zero sobre ele. É por essa razão que a tarefa de relógio sempre verifica quando um temporizador expirou para ver se qualquer um ainda está interessado. A parte difícil de *do_xit* é que o processo pode estar enfileirado ao tentar enviar ou receber no momento em que foi eliminado. O código nas linhas 15056 a 15076 verifica essa possibilidade. Se o processo que está saindo for localizado na fila de mensagens de qualquer outro processo, ele é cuidadosamente removido.

Em contraposição à mensagem anterior, que é ligeiramente complicada, *SYS_GETSP* é absolutamente trivial. Ela

é utilizada pelo gerenciador de memória para obter o valor do ponteiro de pilha atual para algum processo. Esse valor é necessário para as chamadas de sistema BRK e SBRK verem se o segmento de dados e o segmento de pilha colidiram. O código está em *do_getsp* (linha 15089).

Agora chegamos a um dos poucos tipos de mensagem utilizados exclusivamente pelo sistema de arquivos, *SYS_TIMES*. Ela é necessária para implementar a chamada de sistema TIMES, que retorna os tempos de contabilidade para o chamador. Tudo que *do_times* (linha 15106) faz é colocar os tempos solicitados na mensagem de resposta. As chamadas para *lock* e *unlock* são utilizadas para proteger contra uma possível concorrência no acesso aos contadores de tempo.

Pode acontecer que o gerenciador de memória ou o sistema de arquivos descubra um erro que torna impossível continuar a operação. Por exemplo, na inicialização do primeiro, o sistema de arquivos vê que o superbloco no dispositivo raiz foi fatalmente corrompido, ele entra em pane e envia uma mensagem *SYS_ABORT* para o *kernel*. Também é possível o superusuário forçar um retorno para o monitor de inicialização e/ou uma reinicialização utilizando o comando *reboot*, que, por sua vez, executa uma chamada de sistema REBOOT. Em qualquer desses casos, a tarefa de sistema executa *do_abort* (linha 15131), que copia instruções para o monitor, se necessário, e, então, chama *wreboot* para completar o processo.

A maior parte do trabalho de tratamento de sinais é feita pelo gerenciador de memória, que verifica se o processo a ser sinalizado está ativado para detectar ou para ignorar o sinal, se o remetente do sinal estiver intitulado a fazer isso e assim por diante. Entretanto, o gerenciador de memória realmente não pode causar o sinal, o que requer colocar algumas informações na pilha do processo sinalizado.

O tratamento de sinais anterior ao POSIX era problemático, porque capturar um sinal restaurava a resposta, padrão para os sinais. Se o tratamento especial continuado de sinais subseqüentes for requerido, o programador não poderia garantir confiabilidade. Os sinais são assíncronos e um segundo sinal poderia muito bem chegar antes do tratamento ser reativado. O tratamento de sinal no estilo do POSIX resolve esse problema, mas o preço é um mecanismo mais complicado. O tratamento de sinal no antigo estilo poderia ser implementado pelo sistema operacional, empurrando algumas informações para a pilha do processo sinalizado, de maneira semelhante às informações empurradas por uma interrupção. O programador, então, escreveria um manipulador que terminaria com uma instrução de retorno, retirando da pilha as informações necessárias para reassumir a execução. O POSIX salva mais informações, quando um sinal é recebido, do que poderia ser tratado convenientemente dessa maneira. Há um trabalho adicional a fazer mais tarde, antes de o processo sinalizado poder reassumir o que estava fazendo. O gerenciador de memória, assim, precisa enviar duas mensagens para a tarefa de sistema para processar um sinal. A recompensa para esse esforço é um tratamento de sinais mais confiável.

Quando um sinal está para ser enviado para um processo, a mensagem *SYS_SENDSIG* é enviada para a tarefa de sistema. Ela é tratada por *do_sendsig* (linha 15157). As informações necessárias para tratar sinais no estilo POSIX estão em uma estrutura *sigcontext*, que contém o conteúdo dos registradores do processador e uma estrutura *sigframe*, com as informações sobre como sinais são tratados pelo processo. Essas duas estruturas necessitam de alguma inicialização, mas o trabalho básico de *do_sendsig* é simplesmente colocar as informações requeridas na pilha do processo sinalizado e ajustar o contador de programa e o ponteiro da pilha do processo sinalizado de modo que o código de tratamento de sinal seja executado da próxima vez que o agendador der permissão para o processo executar.

Quando um manipulador de sinais no estilo POSIX completa seu trabalho, ele não retira da pilha o endereço onde a execução do processo interrompido é reassumida, como é o caso com sinais no estilo antigo. Ao escrever o manipulador, o programador escreve uma instrução de retorno (ou o equivalente na linguagem de alto nível), mas o tratamento da pilha pela chamada SENDSIG faz com que a instrução de retorno leve a execução de uma chamada de sistema SIGRETURN. O gerenciador de memória, então, envia à tarefa de sistema uma mensagem *SYS_SIGRETURN*. Isso é tratado por *do_sigreturn* (linha 15221), que copia a estrutura *sigcontext* de volta no espaço do *kernel* e, então, restaura os registradores do processo sinalizado. O processo interrompido reassumirá a execução no ponto onde foi interrompido da próxima vez que o agendador permitir que ele execute, mantendo qualquer tratamento de sinal especial que tenha sido previamente configurado.

A chamada de sistema SIGRETURN, diferentemente da maioria das outras discutidas nesta seção, não é requerida pelo POSIX. Ela é uma invenção do MINIX, uma maneira conveniente de iniciar o processamento necessário quando um manipulador de sinal está completo. Os programadores não devem utilizar essa chamada; ela não será reconhecida por outros sistemas operacionais e, de qualquer maneira, não há nenhuma necessidade de referenciá-la explicitamente.

Alguns sinais provêm de dentro da imagem do *kernel* ou são tratados por este antes de irem para o gerenciador de memória. Esses sinais incluem sinais outros que se originam de tarefas, como alarmes da tarefa de relógio ou sinais causados por pressionamentos de teclas detectados pela tarefa de terminal, assim como sinais causados por exceções (como divisão por zero ou instruções ilegais) detectados pela CPU. Os sinais que se originam do sistema de arquivos também são tratados primeiro pelo *kernel*. A mensagem *SYS_KILL* é utilizada pelo sistema de arquivos para solicitar que tal tipo de sinal seja gerado. O nome é talvez um pouco enganoso. Isso não tem nada a ver com o tratamento da chamada de sistema KILL, utilizada por proces-

sos normais para enviar sinais. Essa mensagem é tratada por *do_kill* (linha 15276), que faz a verificação normal para uma origem válida da mensagem e, então, chama *cause_sig* para realmente passar o sinal para o processo. Os sinais originados no *kernel* também são passados por uma chamada para essa função, que inicia sinais enviando uma mensagem *KSIG* para o gerenciador de memória.

Sempre que o gerenciador de memória terminou com um desses sinais tipo *KSIG*, ele envia uma mensagem *SYS_ENDSIG* de volta para a tarefa de sistema. Essa mensagem é tratada por *do_endsig* (linha 15294), que decrementa a contagem dos sinais pendentes, e, se alcançar zero, zera o bit *SIG_PENDING* para o processo sinalizado. Se não houver outros sinalizadores ativados para indicar razões pelas quais o processo não deve ser executável, *lock_ready* é, então, chamada para permitir que o processo execute novamente.

A mensagem *SYS_COPY* é a mais intensamente utilizada. Ela é necessária para permitir que o sistema de arquivos e o gerenciador de memória copiem as informações de e para processos de usuário.

Quando um usuário faz uma chamada READ, o sistema de arquivos verifica seu *cache* para ver se tem o bloco necessário. Se não tiver, ele envia uma mensagem à tarefa de disco apropriada para carregá-lo no *cache*. Então, o sistema de arquivos envia uma mensagem à tarefa de sistema dizendo-lhe para copiar o bloco para o processo de usuário. No pior caso, sete mensagens são necessárias para ler um bloco; no melhor, quatro mensagens são necessárias. Ambos os casos são mostrados na Figura 3-51. Tais mensagens são uma fonte significativa de *overhead* no MINIX e são o preço pago pelo projeto altamente modular

A propósito, nos 8088, que não tinham nenhuma proteção, seria muito fácil trapacear e deixar o sistema de arquivos copiar os dados para o espaço de endereço do chamador, mas isso teria transgredido o princípio do projeto. Qualquer pessoa com acesso a uma máquina assim tão antiga e que estivesse interessada em melhorar o desempenho do MINIX deve ver com cuidado esse mecanismo para avaliar quanto comportamento impróprio pode ser tolerado para o ganho no desempenho. Naturalmente, esse recurso para melhora do desempenho não está disponível em máquinas da classe Pentium com mecanismos de proteção.

O tratamento de uma solicitação *SYS_COPY* é simples e direto. Ele é feito por *do_copy* (linha 15316) e consiste em pouco mais que simplesmente extrair os parâmetros da mensagem e chamar *phys_copy*.

Uma maneira de lidar com alguma ineficiência do mecanismo de passagem de mensagem é empacotar múltiplas solicitações em uma mensagem. A mensagem *SYS_VCOPY* faz isso. O conteúdo dessa mensagem é um ponteiro para um vetor especificando múltiplos blocos a serem copiados entre posições de memória. A função *do_vcopy* (linha 15364) executa um laço, extraindo os endereços de origem e de destino e comprimentos de bloco e chamando *phys_copy* repetidamente até que todas as cópias estejam completas. Isso é semelhante à capacidade de

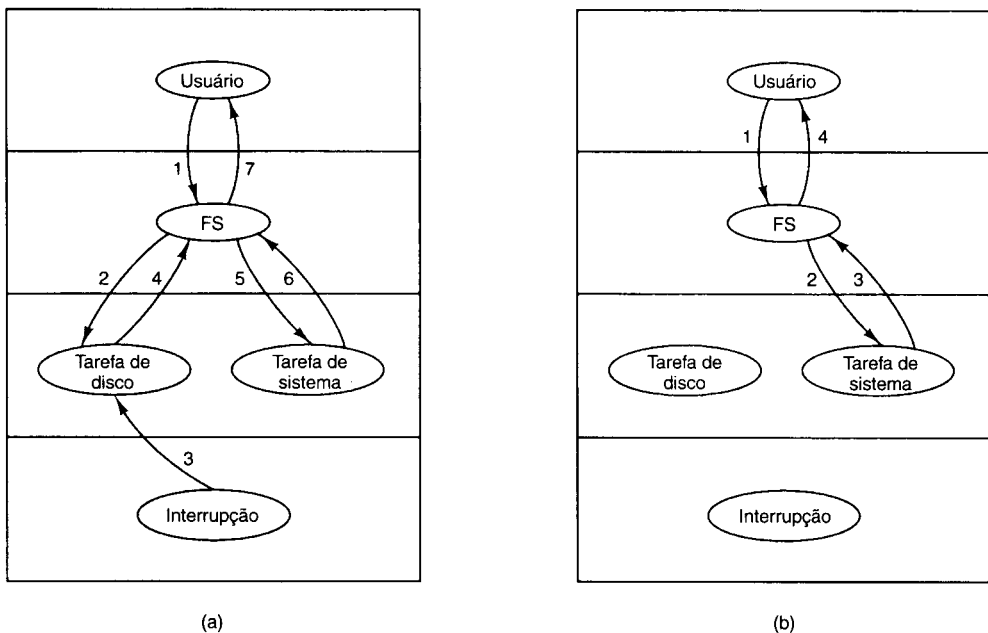



Figura 3-51 (a) O pior caso para ler um bloco exige sete mensagens. (b) O melhor caso para ler um bloco exige quatro mensagens.

dispositivos de disco para tratar múltiplas transferências com base em uma única solicitação.

Há vários outros tipos de mensagem recebida pela tarefa de sistema, a maioria das quais é bem simples. Duas dessas normalmente são utilizadas somente durante a inicialização do sistema. O sistema de arquivos envia uma mensagem *SYS_GBOOT* para solicitar os parâmetros de inicialização. Esses são uma estrutura, *bparam_s*, declarada em *include/minix/boot.h*, que permite que vários aspectos da configuração do sistema sejam especificados pelo programa monitor de inicialização antes de o MINIX ser iniciado. A função *do_gboot* (linha 15403) executa essa operação, que é somente uma cópia de uma parte da memória para outra. Também em tempo de inicialização, o gerenciador de memória envia à tarefa de sistema uma série de mensagens *SYS_MEM* solicitando a base e o tamanho dos trechos de memória disponíveis. *Do_mem* (linha 15424) trata essa solicitação.

A mensagem *SYS_UMAP* é utilizada por um processo que não pertence ao *kernel* para solicitar o cálculo do endereço físico da memória para um endereço virtual dado. *Do_umap* (linha 15445) executa isso chamando *umap*, que é a função chamada no interior do *kernel* para tratar essa conversão.

O último tipo de mensagem que discutiremos é *SYS_TRACE*, que suporta a chamada de sistema PTRACE, utilizada para depuração. Depuração não é uma função fundamental do sistema operacional, mas o suporte do sistema operacional pode torná-la mais fácil. Com a ajuda do sistema operacional, um depurador pode examinar e modificar a memória utilizada por um processo sob teste, assim como o conteúdo dos registradores do processador que são armazenados na tabela de processos sempre que o programa de depuração não está executando.

Normalmente, um processo executa até que bloqueia esperando E/S ou gasta um quantum de tempo. A maioria dos projetos de CPU também oferece uma maneira de limitar a execução de um processo a apenas uma única instrução, ou pode fazer-se com que ele execute somente até que uma instrução particular seja alcançada, colocando um **ponto de interrupção**. Tirar proveito de tais facilidades torna possível uma análise detalhada do programa.

Há 11 operações que podem ser executadas utilizando PTRACE. Algumas são executadas totalmente pelo gerenciador de memória, mas na maioria delas o gerenciador de memória envia uma mensagem *SYS_TRACE* à tarefa de sistema, que, então, chama *do_trace* (linha 15467). Essa função implementa um switch no código de operação de depuração. As operações são, em geral, simples. Um bit *P_STOP* na tabela de processos é utilizado pelo MINIX para reconhecer que a depuração está em progresso e é ligado pelo comando para parar o processo (caso *T_STOP)* ou zerado para reiniciá-lo (caso *T_RESUME)*. A depuração depende do suporte de hardware e, em processadores Intel, é controlada por um bit no registrador de sinalizadores da CPU. Quando o bit é ligado, o processador executa somente uma instrução, então, gera uma exceção *SIGTRAP*. Como já foi mencionado, o gerenciador de memória pára um programa que está sendo depurado quando um sinal é enviado para ele. Esse *TRACEBIT* é tratado pelos comandos *T_STOP* e *T_STEP*. Os pontos de interrupção podem ser configurados de duas maneiras: utilizando o comando *T_SETINS* para substituir uma instrução por um código especial que gera um *SIGTRAP*, ou utilizando o comando *T_SETUSER* para modificar registradores especiais de ponto de interrupção. Em qualquer tipo de sistema para o qual o MINIX pode ser portado, provavelmente será possível implementar um depurador utilizando técnicas similares, mas portar tais funções exigirá estudo do hardware particular.

A maioria dos comandos executados por *dotrace* retorna ou modifica valores no espaço de dados ou de texto do processo em depuração ou em sua entrada da tabela de processos, e o código é simples e direto. Alterar certos registradores e certos bits dos sinalizadores da CPU é algo muito inseguro para ser permitido; então, há muitas verificações no código que trata o comando *T_SETUSER* para evitar essas operações.

No fim de *system.c* estão vários procedimentos utilitários, usados em vários lugares por todo o *kernel*. Quando uma tarefa precisa causar um sinal (p. ex., a tarefa de relógio precisa causar um sinal SIGALRM ou a tarefa de terminal precisa causar um sinal de SIGINT), ela chama *cause_sig* (linha 15586). Esse procedimento liga um bit no campo *p_pending* na entrada da tabela de processos referente ao processo a ser sinalizado e, então, verifica se o gerenciador de memória atualmente está esperando uma mensagem de *ANY*, isto é, se ele está desocupado e esperando a próxima solicitação processar. Se estiver desocupado, *inform* é chamada para dizer para o gerenciador de memória tratar o sinal.

Inform (linha 15627) é chamada somente depois de uma verificação de que o gerenciador de memória não está ocupado, como descrito acima. Além da chamada a partir de *cause_sig*, ela é chamada a partir de *mini_rec* (em *proc.c*) sempre que o gerenciador de memória bloqueia e há sinais pendentes do *kernel*. *Inform* cria uma mensagem do tipo *KSIG* e envia para o gerenciador de memória. A tarefa ou o processo que chama *cause_sig* continua executando logo que a mensagem é copiada para o buffer de recebimento do gerenciador de memória. Ela não espera o gerenciador de memória executar, como seria o caso se o mecanismo de envio normal, que faz o remetente bloquear, fosse utilizado. Antes de retornar, entretanto, *inform* chama *lock_pick_proc*, que agenda o gerenciador de memória para executar. Como as tarefas têm prioridade superior aos servidores, o gerenciador de memória não executa até que todas as tarefas sejam satisfeitas. Quando a tarefa sinalizadora termina, o agendador será iniciado. Se o gerenciador de memória for o processo com prioridade de execução mais alta, ele executará e processará o sinal.

Umap (linha 15658) é um procedimento genérico útil que mapeia um endereço virtual para um endereço físico. Como observamos, ele é chamado por *do_umap*, que serve a mensagem *SYS_UMAP*. Seus parâmetros são um pon-

teiro para a entrada da tabela de processos referente ao processo ou tarefa cujo espaço virtual de endereço está para ser mapeado, um sinalizador que especifica o segmento de texto, dados ou pilha, o próprio endereço virtual e uma contagem de bytes. A contagem de bytes é útil porque *umap* verifica se o buffer inteiro iniciando no endereço virtual está dentro do espaço de endereço do processo. Para fazer isso, ele deve saber o tamanho do buffer. A contagem de bytes não é utilizada para o mapeamento em si, apenas para verificação. Todas as tarefas que copiam dados para ou do espaço do usuário calculam o endereço físico do buffer, utilizando *umap*. Para *drivers* de dispositivo, é conveniente ser capaz de obter os serviços de *umap,* iniciando com o número de processo em vez de um ponteiro para uma entrada da tabela de processos. *Numap* (linha 15697) faz isso. Ele chama *do_newmap* para converter seu primeiro argumento e, então, chama *umap*.

A última função definida em *system.c* é *alloc_segments* (linha 15715). Ela é chamada por *do_newmap*. E também é chamada pela rotina *main* do *kernel* durante a inicialização. Essa definição é muito dependente de hardware. Ela pega as atribuições de segmento que estão registradas em uma entrada da tabela de processos e trata os registradores e os descritores que o processador Pentium utiliza para suportar segmentos protegidos de nível do hardware.

3.11 RESUMO

Entrada/saída é um tema freqüentemente negligenciado, mas importante. Uma parte significativa de qualquer sistema operacional cuida da E/S. Iniciamos vendo o hardware de E/S, e a relação dos dispositivos de E/S com as controladoras de E/S, que são o que o software precisa tratar. Então vimos os quatro níveis de software de E/S: as rotinas de interrupção, os *drivers* de dispositivo, o software de E/S independente de dispositivo e as bibliotecas de E/S, e os *spoolers* que executam no espaço do usuário.

Em seguida, estudamos o problema dos impasses e como ele pode ser abordado. Um impasse ocorre quando, em um grupo de processos, foi concedido a cada um acesso exclusivo a alguns recursos e cada um ainda quer outro recurso que pertence a outro processo no grupo. Todos eles são bloqueados e nenhum jamais executará novamente. O impasse pode ser prevenido estruturando-se o sistema de modo que isso nunca possa ocorrer, por exemplo, permitindo-se que um processo aloque somente um recurso em qualquer instante. O impasse também pode ser evitado examinando-se cada solicitação de recurso para ver se ela conduz a uma situação em que o impasse é possível (um estado inseguro) e negar ou retardar aquelas que levam a problemas.

Os *drivers* de dispositivo no MINIX são implementados como processos embutidos no *kernel*. Vimos o *driver* de disco de RAM, o de disco rígido, o de relógio e o de terminal. A tarefa de alarme síncrono e a tarefa de sistema não são *drivers* de dispositivo mas são estruturalmente muito semelhantes a um. Cada uma dessas tarefas tem um laço principal que recebe solicitações, processa-as e, por fim, envia de volta respostas para informar o que aconteceu. Todas as tarefas estão localizadas no mesmo espaço de endereço. As tarefas de disco de RAM, de disco rígido e de *driver* de disquete utilizam uma única cópia do mesmo laço principal e também compartilham funções comuns. Contudo, cada uma é um processo independente. Vários terminais diferentes, utilizando o console do sistema, as linhas seriais e as conexões de rede são suportados por uma única tarefa de terminal.

Os *drivers* de dispositivo têm relacionamentos variáveis com o sistema de interrupções. Os dispositivos que podem completar seu trabalho rapidamente, como o disco de RAM e o dispositivo de vídeo mapeado em memória, não utilizam interrupções de modo algum. A tarefa do *driver* de disco rígido faz a maior parte do seu trabalho no próprio código da tarefa, e os manipuladores de interrupções simplesmente informam o status de retorno. O próprio manipulador de interrupções de relógio faz diversas operações de contabilidade e somente envia uma mensagem para a tarefa de relógio quando há algum trabalho que não pode ser feito pelo manipulador. O manipulador de interrupções de teclado *bufferiza* a entrada e nunca envia uma mensagem para sua tarefa. Em vez disso, ele muda uma variável inspecionada pelo manipulador de interrupções de relógio; este último envia uma mensagem para a tarefa de terminal.

EXERCÍCIOS

1. Imagine que os avanços na tecnologia dos chips tornem possível colocar uma controladora inteira, incluindo toda a lógica de acesso de barramento, em um chip de baixo custo. Como isso afetará o modelo da Figura 3-1?

2. Se uma controladora de disco grava os bytes que recebe da memória de disco tão rapidamente quanto ela os recebe, sem nenhuma bufferização interna, a intercalação é concebivelmente útil? Discuta.

3. Com base na velocidade de rotação e na geometria dos discos, quais são as taxas de bit para transferências entre o próprio disco e o buffer da controladora para um disquete e para um disco rígido? Como isso se compara com outras formas de E/S (linhas seriais e redes)?

4. Um disco é duplamente intercalado, como na Figura 3-4(c). Ele tem oito setores de 512 bytes por trilha e uma velocidade de rotação de 300rpm. Quanto tempo ele leva para ler todos

os setores de uma trilha em ordem, supondo que o braço já esteja corretamente posicionado e 1/2 rotação seja necessária para obter o setor 0 sob o cabeçote? O que é a taxa de transmissão de dados? Agora repita o problema para um disco não-intercalado com as mesmas características. Quanto a taxa de transmissão de dados degrada devido à intercalação?

5. O multiplexador de terminal DM-11, que foi utilizado no PDP-11, há muitos e muitos anos, fazia uma amostragem de cada linha (*half duplex*) de terminal a sete vezes a taxa de transmissão de dados (*baud rate*) para ver se o bit de entrada era um 0 ou um 1. A amostragem da linha levava 5,7 microssegundos. Quantas linhas de 1.200 *bauds* o DM-11 podia suportar?

6. Uma rede local é utilizada como segue. O usuário emite uma chamada de sistema para gravar pacotes de dados na rede. O sistema operacional, então, copia os dados para um buffer do *kernel*. Então, ele copia os dados para a controladora de rede. Quando todos os bytes estão seguramente dentro da controladora, eles são enviados pela rede a uma taxa de 10 megabits/s. A controladora de rede receptora armazena cada bit em um microssegundo depois de ele ser enviado. Quando o último bit chega, a CPU de destino sofre uma interrupção. O *kernel* copia o pacote recém-chegado para um buffer do *kernel* para inspecioná-lo. Uma vez que tenha descoberto o usuário a quem o pacote destina-se, o *kernel* copia os dados para o espaço do usuário. Se supormos que cada interrupção e seu processamento associado leva 1ms, que os pacotes são de 1024 bytes (ignorando os cabeçalhos) e que copiar um byte leva 1 microsegundo, qual será a taxa máxima em que um processo pode transmitir dados para o outro? Suponha que o remetente esteja bloqueado até que o trabalho tenha terminado no lado do receptor e que um aviso de reconhecimento seja retornado. Para simplificar, suponha que o tempo para obter o aviso de reconhecimento de volta é tão pequeno que pode ser ignorado.

7. O que é "independência de dispositivo"?

8. Em qual das quatro camadas de software de E/S é feito cada uma das seguintes operações:
   (a) Calcular a trilha, o setor e o cabeçote para uma leitura de disco
   (b) Manter um *cache* de blocos recentemente utilizados.
   (c) Gravar comandos nos registradores do dispositivo
   (d) Verificar se o usuário tem permissão para utilizar o dispositivo.
   (e) Converter inteiros binários em ASCII para impressão.

9. Por que arquivos de saída para a impressora normalmente sofrem *spool* em disco antes de serem impressos?

10. Considere a Figura 3-8. Suponha que no passo (o) C tenha solicitado S em vez de R. Isso levaria a um impasse? E supondo que ele tenha solicitado tanto S quanto R?

11. Faça um cuidadoso exame da Figura 3-11(b). Se Suzanne solicitar mais uma unidade, isso levaria a um estado seguro ou a um estado inseguro? E se a solicitação viesse de Marvin em vez de Suzanne?

12. Todas as trajetórias na Figura 3-12 são horizontais ou verticais. Você pode vislumbrar qualquer circunstância em que trajetórias diagonais também sejam possíveis?

13. Suponha que o processo na Figura 3-13 solicite a última unidade de fita. Essa ação conduz a um impasse?

14. Um computador tem seis unidades de fita, com n processos competindo por elas. Cada processo pode solicitar duas unidades. Para quais valores de n o sistema ficaria livre de impasses?

15. Um sistema pode estar em um estado que não seja nem um estado de impasse nem um estado seguro? Se pode, dê um exemplo. Se não, prove que todos os estados são ou um estado de impasse ou um estado seguro.

16. Um sistema distribuído que utiliza caixas de correio tem duas primitivas IPC, SEND e RECEIVE. A última primitiva especifica um processo do qual deve receber e bloqueia se nenhuma mensagem desse processo estiver disponível, mesmo que possam estar sendo esperadas mensagens de outros processos. Não há recursos compartilhados, mas os processos precisam comunicar-se freqüentemente sobre outros assuntos. O impasse é possível? Discuta.

17. Em um sistema de transferência eletrônica de fundos, há centenas de processos idênticos que trabalham como segue. Cada processo lê uma linha de entrada especificando uma quantidade de dinheiro, a conta a ser creditada e a conta a ser debitada. Então, ele bloqueia ambas as contas e transfere o dinheiro, liberando os bloqueios quando tiver terminado. Com muitos processos executando em paralelo, há um perigo muito real de que tendo bloqueado a conta x ele será incapaz de bloquear y porque y foi bloqueado por um processo agora esperando x. Esboce um esquema que evite impasses. Não libere um registro de conta até que você tenha completado as transações. (Em outras palavras, não são permitidas soluções que bloqueiam uma conta e, então, liberam-na imediatamente se a outra estiver bloqueada).

18. O algoritmo do banqueiro está sendo executado em um sistema com m classes de recursos e n processos. No limite de grandes m e n, o número de operações que devem ser executadas para verificar a segurança de um estado é proporcional a $m^a n^b$. Quais são os valores de a e b ?

19. A Cinderela e o Príncipe estão divorciando-se. Para dividir sua propriedade, eles concordaram no seguinte algoritmo. Toda manhã, cada um deles pode enviar uma carta para o advogado do outro solicitando um item das propriedades. Como leva um dia para as cartas serem entregues, eles concordaram que se ambos descobrirem que pediram o mesmo item no mesmo dia, no dia seguinte eles enviarão uma carta cancelando a solicitação. Entre suas propriedades, estão seu cão, Woofer, a casinha de Woofer, seu canário, Tweeter, e a gaiola deste último. Os animais amam suas casas, então, foi concordado que qualquer divisão de propriedade que separasse um animal de sua casa seria inválida, exigindo que a divisão recomeçasse do zero. Tanto o Príncipe como a Cinderela querem desesperadamente seu cão Woofer. Assim, eles saem em férias (separados), tendo cada um programado um computador pessoal para tratar a negociação. Quando eles voltam das férias, os computadores ainda estão negociando. Por quê? O impasse é possível? É possível ocorrer fome (esperar eternamente)? Discuta.

20. O formato da mensagem da Figura 3-15 é utilizado para enviar mensagens de solicitação para *drivers* de dispositivos de bloco. Que campos poderiam ser omitidos, se é que

algum poderia ser omitido, em mensagens para dispositivos de caractere?

21. Chegam ao *driver* de disco solicitações pelos cilindros 10, 22, 20, 2, 40, 6 e 38, nessa ordem. Uma busca leva 6ms por cilindro movido. Quanto tempo de busca é necessário para:
    (a) Primeiro a entrar, primeiro a ser servido.
    (b) Cilindro mais próximo em seguida
    (c) Algoritmo do elevador (inicialmente movendo-se para cima).

    Em todos os casos, o braço está inicialmente no cilindro 20.

22. Um vendedor de computadores pessoais, em visita a uma universidade no sudoeste de Amsterdã, alardeava ao abordar um cliente que sua companhia tinha dedicado esforço substancial para tornar sua versão do UNIX mais veloz. Como exemplo, ele apontou que seu *driver* de disco utilizava o algoritmo do elevador e também enfileirava múltiplas solicitações dentro de um cilindro pela ordem dos setores. Um aluno, Harry Hacker, ficou impressionado e comprou um. Levou-o para casa e escreveu um programa para ler aleatoriamente 10.000 blocos ao longo do disco. Para sua surpresa, o desempenho que ele mediu era idêntico ao que seria esperado do algoritmo primeiro a entrar, primeiro a ser servido. O vendedor estava mentindo?

23. Um processo UNIX tem duas partes — a parte do usuário e a parte do *kernel*. A parte do *kernel* é como uma sub-rotina ou uma co-rotina?

24. O manipulador de interrupções de relógio em um certo computador exige 2ms (incluindo o *overhead* da comutação de processos) por tique de relógio. O relógio executa em 60Hz. Que fração da CPU é dedicada ao relógio?

25. Dois exemplos de temporizadores *watchdog* foram dados no texto: sincronização da inicialização do motor de disquete e permissão para retorno de carro em terminais de impressão. Dê um terceiro exemplo.

26. Por que os terminais RS-232 são baseados em interrupções, mas os terminais mapeados em memória não são?

27. Considere o modo como um terminal funciona. O *driver* dá saída a um caractere e, então, bloqueia. Quando o caractere é impresso, uma interrupção ocorre, e uma mensagem é enviada para o *driver* bloqueado, que dá saída para o próximo caractere e, então, bloqueia novamente. Se o tempo de passagem de uma mensagem, saída de um caractere e bloqueio é 4ms, esses métodos funcionam bem em linhas de 110 *bauds*? E em linhas de 4800 *bauds*?

28. Um terminal de mapa de bits contém 1.200 por 800 *pixels*. Para rolar uma janela, a CPU (ou controladora) deve mover todas as linhas de texto para cima copiando seus bits de uma parte da RAM de vídeo para outra. Se uma janela em particular tiver 66 linhas de altura por 80 caracteres (5280 caracteres, no total) e a caixa de caractere tiver 8 *pixels* de largura por 12 *pixels* de altura, quanto tempo é necessário para rolar a janela inteira a uma velocidade de cópia de 500ns por byte? Se todas as linhas tiverem 80 caracteres de comprimento, qual é a taxa de transmissão de dados (*baud rate*) equivalente do terminal? Colocar um caractere na tela leva 50ms. Agora calcule a taxa de transmissão de dados para o mesmo terminal colorido, com 4 bits/*pixel*. (Colocar um caractere na tela agora toma 200ms.)

29. Por quê os sistemas operacionais oferecem caracteres de escape como CTRL-V no MINIX?

30. Depois de receber um caractere DEL (SIGINT), o *driver* do MINIX descarta toda a saída atualmente enfileirada para o terminal envolvido. Por quê?

31. Muitos terminais RS-232 têm seqüências de escape para excluir a linha atual e mover para cima uma linha todas as linhas embaixo da linha atual. Como você acha que esse recurso está implementado dentro do terminal?

32. No monitor colorido original do IBM PC, escrever na RAM de vídeo em qualquer momento que não durante o retraço vertical do feixe do CRT causava manchas que apareciam por toda a tela. Uma imagem de tela tem 25 por 80 caracteres, cada um dos quais se ajusta em uma caixa de 8 × 8 *pixels*. Cada linha de 640 *pixels* é desenhada em uma única varredura horizontal do feixe, o que leva 63,6ms, incluindo o retraço horizontal. A tela é redesenhada 60 vezes por segundo, cada uma das quais requer o período de um retraço vertical para fazer o feixe voltar ao topo. Que fração de tempo dispõe a RAM de vídeo para gravar?

33. Escreva um *driver* gráfico para o monitor IBM colorido, ou algum outro monitor de mapa de bits conveniente. O *driver* deve aceitar comandos para ligar e para limpar *pixels* individuais, para mover retângulos pela tela e para quaisquer outros recursos que você acredita serem interessantes. Programas do usuário fazem interface com o *driver*, abrindo */dev/graphics* e gravando comandos aí.

34. Modifique o *driver* de disquete do MINIX para fazer *cache* de uma trilha por vez.

35. Implemente um *driver* de disquete que funcione como um dispositivo de caractere, em vez de bloco, para pular o *cache* de blocos do sistema de arquivos. Dessa maneira, os usuários podem ler grandes trechos de dados do disco, que são "DMAdos" diretamente para espaço do usuário, favorecendo significativamente o desempenho. Esse *driver* seria de interesse principalmente para programas que precisam ler os bits brutos no disco, sem considerar o sistema de arquivos. Os verificadores dos sistemas de arquivos entram nessa categoria.

36. Implemente a chamada de sistema PROFIL do UNIX, que está faltando no MINIX.

37. Modifique o *driver* de terminal de modo que além de ter uma tecla especial para apagar o caractere anterior, haja uma tecla para apagar a palavra anterior.

38. Um novo dispositivo de disco rígido com mídia removível foi adicionado a um sistema MINIX. Esse dispositivo deve alcançar a velocidade de rotação toda vez que as mídias são trocadas, e o laço para isso é bem longo. Já se sabe que as trocas de mídia serão feitas com freqüência enquanto o sistema estiver executando. De repente, a rotina *waitfor* em *at_wini.c* torna-se insatisfatória. Projete uma nova rotina *waitfor* em que, se o padrão de bits sendo esperado não for encontrado depois de 1 segundo de espera ativa, o código entrará em uma fase em que a tarefa de disco irá dormir por 1 segundo, testar a porta e voltar a dormir durante outro segundo até que o padrão buscado seja encontrado ou o período predefinido *TIMEOUT* expire.

4

Gerenciamento de Memória

A memória é um recurso importante que deve ser gerenciado com cuidado. Enquanto o computador doméstico médio hoje em dia tem 50 vezes mais memória do que o IBM 7034, o maior computador no mundo no início da década de 60, os programas estão crescendo tão rapidamente quanto as memórias. Parafraseando a lei de Parkinson,* poderíamos dizer que "os programas tendem a expandir até ocupar toda a memória disponível para armazená-los". Neste capítulo, estudaremos como o sistema operacional gerencia a memória.

Idealmente, o que todo programador gostaria de ter à disposição é uma memória rápida e infinitamente grande que também fosse não-volátil, isto é, que não perdesse seu conteúdo quando a energia elétrica cai. E já que estamos divagando, por que também não pedir para que fosse bem barata? Infelizmente, a tecnologia ainda não oferece esse tipo de memória. A maioria dos computadores, portanto, tem uma **hierarquia de memória**, com uma pequena quantidade de memória de *cache* volátil, muito rápida e cara, alguns megabytes de memória principal volátil (RAM), de velocidade e preço médios, além de centenas ou milhares de megabytes de armazenamento em disco, não-volátil, lento e barato. O trabalho do sistema operacional é coordenar como essas memórias são utilizadas.

A parte do sistema operacional que gerencia a hierarquia de memória é chamada **gerenciador de memória.** Seu trabalho é controlar que partes da memória estão em uso e que partes não estão, alocar memória para processos quando eles necessitarem e desalocar quando eles terminarem, e gerenciar a troca entre a memória principal e o disco quando a memória principal é muito pequena para armazenar todos os processos.

Neste capítulo, investigaremos diversos esquemas de gerenciamento de memória, variando dos muito simples aos altamente sofisticados. Começaremos do princípio e veremos primeiro o sistema de gerenciamento de memória mais simples possível e, então, gradualmente avançaremos para aqueles cada vez mais elaborados.

4.1 GERENCIAMENTO BÁSICO DE MEMÓRIA

Os sistemas de gerenciamento de memória podem ser divididos em duas classes: aqueles que movem processos de um lado para outro entre a memória principal e o disco durante execução (fazendo troca e paginação) e aqueles que não o fazem. Os últimos são mais simples; então, vamos estudá-los primeiro. Mais adiante neste capítulo, examinaremos troca e paginação. Ao longo de todo este capítulo, o leitor deve manter em mente que troca e paginação são em grande parte artefatos causados pela falta de memória principal suficiente para armazenar todos os programas simultaneamente. À medida que a memória principal fica mais barata, os argumentos a favor de um tipo de esquema de gerenciamento de memória ou outro podem tornar-se obsoletos — a menos que os programas aumentem mais rapidamente do que o preço da memória diminui.

*N. de T. Observações satíricas propostas como leis econômicas, especialmente "o trabalho tende a aumentar até ocupar todo o tempo disponível para sua conclusão". O nome provém do seu autor, o historiador britânico Cyril Northcote Parkinson (1909-1993), famoso por seus trabalhos humorísticos que ridicularizavam a ineficiência das burocracias.

4.1.1 Monoprogramação sem Troca ou Paginação

O esquema mais simples possível de gerenciamento de memória é executar somente um programa por vez, compartilhando a memória entre esse programa e o sistema operacional. Três variações desse tema são mostradas na Figura 4-1. O sistema operacional pode estar na parte inferior da memória em RAM (*Random Access Memory*, Memória de Acesso Aleatório), como mostrado na Figura 4-1(a), ou pode estar em ROM (*Read-Only Memory*, Memória Apenas de Leitura) na parte superior da memória, como mostrado na Figura 4-1(b), ou os *drivers* de dispositivo podem estar na parte superior da memória em uma ROM e o restante do sistema em RAM na parte inferior, como mostrado na Figura 4-1(c). O último modelo é utilizado por pequenos sistemas MS-DOS, por exemplo. No IBM PC, a parte do sistema na ROM é chamada BIOS (*Basic Input Output System*, Sistema Básico de Entrada e Saída).

Quando o sistema está organizado dessa maneira, somente um processo por vez pode estar executando. Logo que o usuário digita um comando, o sistema operacional copia o programa solicitado do disco para a memória e executa-o. Quando o processo termina, o sistema operacional exibe um aviso de comando e espera um novo comando. Quando recebe o comando, ele carrega um novo programa na memória, sobrepondo o primeiro

4.1.2 Multiprogramação com Partições Fixas

Embora a monoprogramação seja, às vezes, utilizada em computadores pequenos com sistema operacional simples, com freqüência é desejável permitir que múltiplos processos executem simultaneamente. Em sistemas de compartilhamento de tempo, ter múltiplos processos na memória simultaneamente significa que quando um processo é bloqueado esperando a E/S acabar, outro pode utilizar a CPU. Assim, a multiprogramação aumenta a utilização da CPU. Entretanto, mesmo em computadores pessoais é freqüentemente útil ser capaz de executar dois ou mais programas ao mesmo tempo.

A maneira mais fácil de alcançar a multiprogramação é simplesmente dividir a memória em n partições (possivelmente desiguais). Esse particionamento pode ser feito, por exemplo, manualmente quando o sistema é iniciado.

Quando um *job* chega, pode ser colocado na fila de entrada da menor partição capaz de armazená-lo. Uma vez que as partições são fixas nesse esquema, qualquer espaço em uma partição não-utilizada por um *job* é perdido. A Figura 4-2(a) apresenta um esquema desse sistema de partições fixas e de filas de entrada separadas.

A desvantagem de classificar os *jobs* de entrada em filas separadas torna-se aparente quando a fila para uma partição grande está vazia, mas a fila para uma partição pequena está cheia, como é o caso para as partições 1 e 3 na Figura 4-2(a). Uma organização alternativa é manter uma fila única como na Figura 4-2(b). Sempre que uma partição torna-se livre, o *job* mais próximo do início da fila que se ajusta na partição vazia poderia ser carregado nela e executado. Uma vez que é indesejável desperdiçar uma partição grande com um *job* pequeno, uma estratégia diferente é pesquisar a fila de entrada inteira sempre que uma partição tornar-se livre e selecionar o maior *job* que se ajusta. Note que o último algoritmo discrimina *jobs* pequenos porque estes não merecem ter uma partição inteira, ao passo que normalmente é desejável dar o melhor serviço aos *jobs* pequenos (que supostamente são interativos), não o pior.

Uma saída é dispor de pelo menos uma partição pequena, a qual permitirá que *jobs* pequenos sejam executados sem alocar uma partição grande para eles.

Outra abordagem é estabelecer uma regra determinando que um *job* elegível para executar não pode ser ignora-

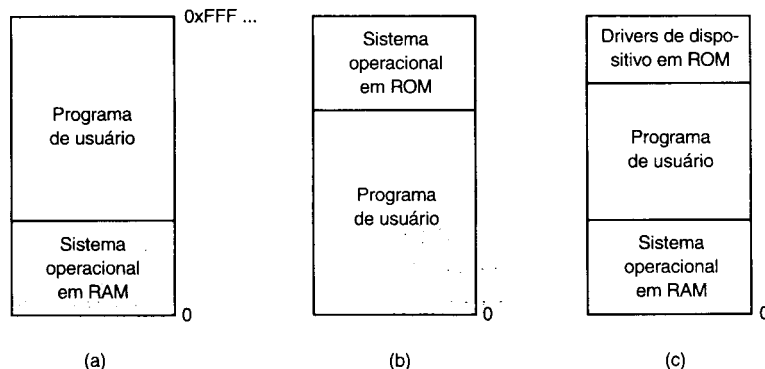



Figura 4-1 Três maneiras simples de organizar memória com um sistema operacional e com um processo de usuário. Também existem outras possibilidades.

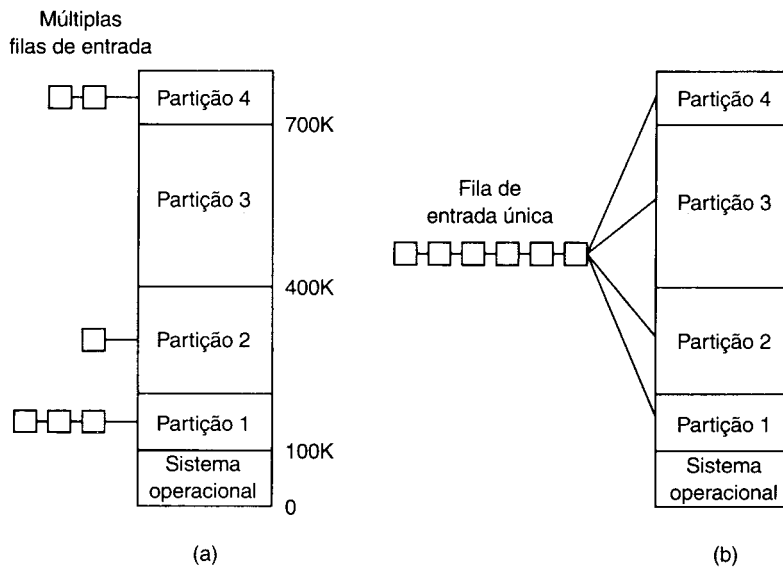



Figura 4-2 (a) Partições fixas de memória com filas separadas para cada partição. (b) Partições fixas de memória com uma única fila de entrada.

do mais que *k* vezes. Cada vez que é ignorado, ele ganha um ponto. Quando adquiriu *k* pontos, ele não pode ser ignorado novamente.

Esse sistema, com partições fixas definidas pela manhã pelo operador e não alteradas depois, foi utilizado por *mainframes* IBM OS/360 de grande porte durante muitos anos. Ele foi chamado **MFT** (Multiprogramação com um número Fixo de Tarefas ou OS/MFT). Ele é simples de entender e igualmente simples de implementar: os *jobs* que chegam são enfileirados até que uma partição adequada esteja disponível, momento em que o *job* é carregado nessa partição e executa até a sua conclusão. Hoje em dia, poucos sistemas operacionais suportam esse modelo, se é que algum suporta.

Realocação e Proteção

A multiprogramação introduz dois problemas essenciais que devem ser resolvidos — realocação e proteção. Olhe na Figura 4-2. Na figura está claro que *jobs* diferentes serão executados em endereços também diferentes. Quando um programa é vinculado (i. e., o programa principal, procedimentos de usuário gravado, e procedimentos de biblioteca são combinados em um único espaço de endereço), o *linkeditor* deve saber em que endereço o programa deve começar na memória.

Por exemplo, suponha que a primeira instrução seja uma chamada para um procedimento no endereço absoluto 100 dentro do arquivo binário, produzido pelo *linkeditor*. Se esse programa for carregado na partição 1, essa instrução saltará para o endereço absoluto 100, que está dentro do sistema operacional. O que é necessário é uma chamada para 100K + 100. Se o programa for carregado na partição 2, ele executará como uma chamada para 200K + 100 e assim por diante. Esse problema é conhecido como problema da **realocação**.

Uma possível solução é realmente modificar as instruções enquanto o programa é carregado na memória. O programa carregado na partição 1 tem 100K adicionado a cada endereço, o programa carregado na partição 2 tem 200K adicionado aos endereços e assim por diante. Para realizar uma realocação como essa durante o carregamento, o *linkeditor* deve incluir no programa binário uma lista ou um mapa de bits, informando quais palavras do programa são endereços a serem realocados e quais são códigos de instruções, constantes ou outros itens que não devem ser realocados. O OS/MFT funcionava dessa maneira. Alguns microcomputadores também trabalham assim.

A realocação durante o carregamento não resolve o problema da proteção. Um programa malicioso sempre pode construir uma nova instrução e saltar para ela. Como os programas nesse sistema utilizam endereços absolutos de memória em vez de endereços relativos a um registrador, não há como impedir que um programa crie uma instrução que lê ou grava qualquer palavra na memória. Em sistemas multiusuários, é indesejável deixar um processo ler ou gravar memória pertencente a outros usuários.

A solução que a IBM escolheu para proteger os 360 foi dividir a memória em blocos de 2KB e atribuir um código de proteção de 4 bits a cada bloco. O PSW continha uma chave de 4 bits. O hardware do 360 interrompia qualquer tentativa, por parte de um processo em execução, de acessar memória cujo código de proteção diferia da chave PSW. Uma vez que somente o sistema operacional podia alterar

os códigos e a chave de proteção, os processos de usuário eram impedidos de interferir um no outro e no próprio sistema operacional.

Uma solução alternativa para ambos os problemas, realocação e proteção, é equipar a máquina com dois registradores especiais de hardware, chamados **registrador de base** e **registrador de limite**. Quando um processo é agendado, o registrador de base é carregado com o endereço do início de sua partição, e o registrador de limite é carregado com o comprimento de sua partição. Todo endereço de memória gerado tem o registrador de base automaticamente adicionado a ele próprio antes de ser enviado para memória. Assim, se o registrador de base for 100K, uma instrução CALL 100 efetivamente transforma-se em uma instrução CALL 100K + 100, sem que a instrução em si seja modificada. Os endereços também são verificados em relação ao registrador de limite para certificar-se de que eles não tentarão endereçar memória fora da partição atual. O hardware protege os registradores de base e de limite para impedir que os programas de usuário os modifiquem.

O CDC 6600 — o primeiro supercomputador do mundo — utilizava esse esquema. A CPU Intel 8088 utilizada no IBM PC original empregava uma versão mais fraca desse esquema — os registradores de base, mas não os registradores de limite. Desde os 286, um esquema melhor foi adotado.

4.2 TROCA

Com um sistema de lotes, organizar a memória em partições fixas é simples e efetivo. Cada *job* é carregado em uma partição quando chega no começo da fila e permanece na memória até que termine. Contanto que *jobs* suficientes possam ser mantidos na memória para manter a CPU ocupada todo o tempo, não há nenhuma razão para utilizar qualquer coisa mais complicada.

Com sistemas de compartilhamento de tempo ou computadores gráficos pessoais, a situação é diferente. Às vezes, não há memória principal suficiente para armazenar todos os processos atualmente ativos, então, os processos em excesso são mantidos no disco e trazidos de lá para execução dinamicamente.

Duas abordagens gerais para o gerenciamento de memória podem ser utilizadas, dependendo (em parte) do hardware disponível. A estratégia mais simples, chamada **troca**, consiste em trazer cada processo inteiro, executá-lo temporariamente e, então, devolvê-lo ao disco. A outra estratégia, chamada **memória virtual**, permite que os programas executem mesmo quando estão apenas parcialmente na memória principal. A seguir, estudaremos o processo de troca; na Seção 4-3, examinaremos a memória virtual.

A operação de um sistema de troca é ilustrada na Figura 4-3. Inicialmente somente um processo está na memória. Então, os processos B e C são criados ou recuperados do disco. Na Figura 4-3(d), *A* terminou ou é enviado para o disco. Então, *D* entra, e *B* sai. Por fim, *E* entra.

A principal diferença entre as partições fixas da Figura 4-2 e as partições variáveis da Figura 4-3 é que o número, a posição e o tamanho das partições variam dinamicamente na última, enquanto os processos entram e saem, ao passo que são fixos na primeira. A flexibilidade de não se prender a um número fixo de partições que podem ser ou muito grandes ou muito pequenas otimiza a utilização da memória, mas também complica a tarefa de alocar e desalocar a memória, assim como a monitoração da memória utilizada.

Quando a troca cria múltiplas lacunas na memória, é possível juntar todas elas em um grande espaço, movendo todos os processos para baixo o máximo possível. Essa téc-

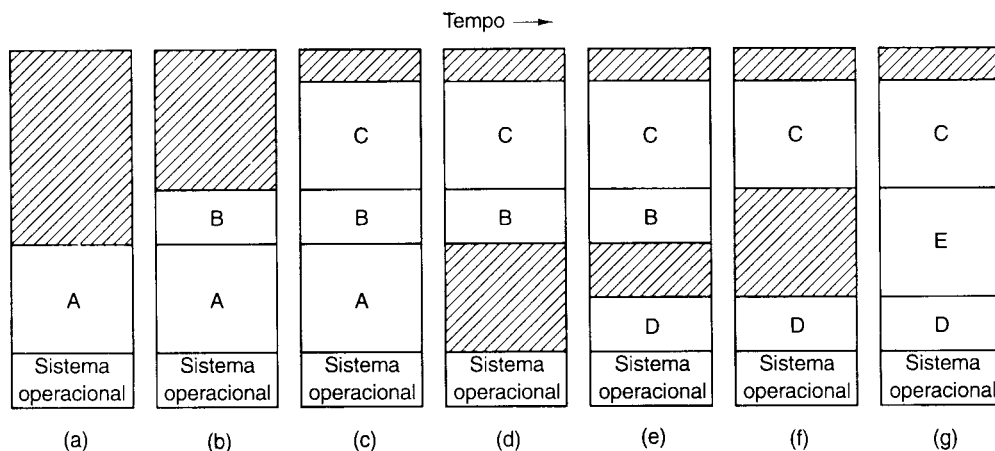



Figura 4-3 As alterações de alocação de memória enquanto os processos entram e saem da memória. As regiões sombreadas correspondem à memória não-utilizada.

nica é conhecida como **compactação de memória.** Normalmente ela não é feita porque exige muito tempo da CPU. Por exemplo, uma máquina com 32MB que pode copiar 16 bytes por microssegundo, leva 2 s para compactar toda a memória.

Um ponto que merece ser considerado é quanta memória deve ser alocada para um processo quando ele é criado ou é recuperado do disco. Se processos são criados com um tamanho fixo que nunca muda, então, a alocação é simples: aloca-se exatamente o que é requerido, nem mais nem menos.

Se, porém, os segmentos de dados dos processos podem crescer, por exemplo, alocando memória dinamicamente a partir de um *heap*, como em muitas linguagens de programação, ocorre um problema sempre que um processo tenta crescer. Se existe uma lacuna adjacente ao processo, ela pode ser alocada e oferecida ao processo. Por outro lado, se o processo é adjacente a outro processo, o processo em crescimento terá de ser movido para uma lacuna de memória suficientemente grande para ele ou um ou mais processos terão de ser enviados para o disco para criar uma lacuna suficientemente grande. Se um processo não pode crescer na memória, e a área de troca no disco está cheia, o processo deverá esperar ou ser eliminado.

Se é esperado que a maioria dos processos crescerá ao executar, provavelmente é uma boa idéia alocar uma pequena memória extra sempre que se fizer a troca ou mover-se um processo, reduzindo o *overhead* associado com mover ou com trocar processos que não cabem mais na memória alocada para eles. Entretanto, ao enviar processos para o disco, só a memória realmente em uso deve ser enviada; é um desperdício enviar a memória extra também. Na Figura 4-4(a) vemos uma configuração de memória em que o espaço para crescimento foi alocado para dois processos.

Se os processos têm dois segmentos que podem crescer, por exemplo, o segmento de dados sendo utilizado como um *heap* para variáveis que são dinamicamente alocadas e liberadas e um segmento de pilha para as variáveis locais normais e para os endereços de retorno, um arranjo alternativo sugere o que é mostrado na Figura 4-4(b). Nessa figura, vemos que cada processo ilustrado tem uma pilha na parte superior de sua memória alocada que está crescendo para baixo e um segmento de dados imediatamente após o texto do programa que está crescendo para cima. A memória entre eles pode ser utilizada por qualquer um dos segmentos. Se ele precisar de mais, ou o processo precisará ser movido para uma lacuna com espaço suficiente, fazendo troca da memória com disco até que uma lacuna seja suficientemente grande possa ser criada, ou será eliminado.

4.2.1 Gerenciamento de Memória com Mapas de Bits

Quando a memória é atribuída dinamicamente, o sistema operacional deve gerenciá-la. Em termos gerais, há duas maneiras de monitorar o uso da memória: mapas de bits e listas livres. Nesta e na próxima seção, veremos esses dois métodos.

Com um mapa de bits, a memória é dividida em unidades de alocação, talvez tão pequenas quanto algumas palavras e talvez tão grandes quanto vários kilobytes. Cor-

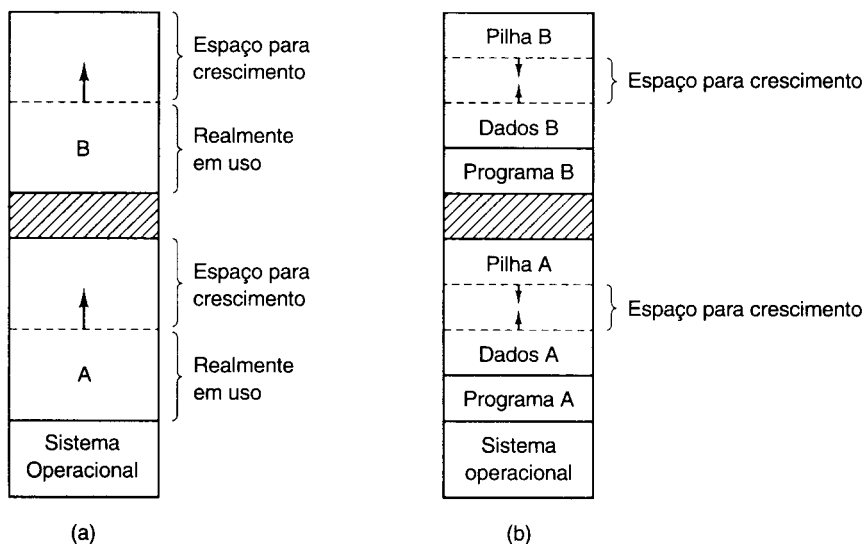



Figura 4-4 (a) Alocando espaço para um segmento de dados crescente. (b) Alocando espaço para uma pilha e para um segmento de dados crescente.

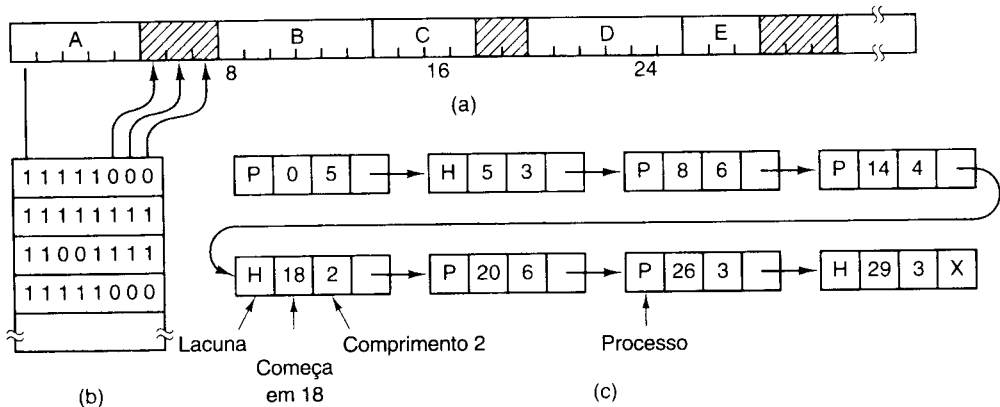



Figura 4-5 (a) Uma parte da memória com cinco processos e três lacunas. Os traços menores mostram as unidades de alocação da memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) O mapa de bits correspondente. (C) As mesmas informações como uma lista.

respondendo a cada unidade de alocação há um bit no mapa de bits, que é 0 se a unidade está livre, e 1 se estiver ocupada (ou vice-versa). A Figura 4-5 mostra parte da memória e o mapa de bits correspondente.

O tamanho da unidade de alocação é uma importante questão de projeto. Quanto menor a unidade de alocação, maior o mapa de bits. Entretanto, mesmo com uma unidade de alocação tão pequena quanto 4 bytes, 32 bits de memória exigirão somente 1 bit do mapa. Uma memória de 32n bits utilizará n bits de mapa, portanto o mapa de bits ocupará apenas 1/33 da memória. Se a unidade de alocação escolhida for grande, o mapa de bits será menor, mas uma quantia apreciável de memória pode ser desperdiçada na última unidade se o tamanho do processo não for um múltiplo exato da unidade de alocação.

Um mapa de bits oferece uma maneira simples de monitorar palavras de memória em uma quantidade fixa de memória porque o tamanho do mapa de bits depende somente do tamanho da memória e do tamanho da unidade de alocação. O problema principal com isso é que quando se decide trazer um processo de k unidades para a memória, o gerenciador de memória deve pesquisar no mapa de bits para encontrar uma lacuna de k bits 0 consecutivos no mapa. Localizar em um mapa de bits uma lacuna de um comprimento dado é uma operação lenta (porque a lacuna pode escarranchar-se na faixa de palavras no mapa); este é um argumento contra os mapas de bits.

4.2.2 Gerenciamento de Memória com Listas Encadeadas

Outra maneira de monitorar a memória é manter uma lista encadeada dos segmentos de memória alocados e livres, onde um segmento é um processo ou uma lacuna entre dois processos. A memória da Figura 4-5(a) é representada na Figura 4-5(c) como uma lista encadeada de segmentos. Cada entrada na lista especifica uma lacuna (H) ou processo (P), o endereço em que inicia, o comprimento e um ponteiro para a próxima entrada.

Nesse exemplo, a lista de segmentos está classificada por endereço. Classificar dessa maneira tem a vantagem de que quando um processo termina ou está sendo enviado para disco, atualizar a lista é simples e direto. Um processo que termina normalmente tem dois vizinhos (exceto quando está na parte superior da memória). Esses podem ser tanto processos como lacunas, o que leva às quatro combinações da Figura 4-6. Na Figura 4-6(a), a atualização da lista exige substituir um P por um H. Na Figura 4-6(b) e na Figura 4-6(c), duas entradas são fundidas em uma, e a lista torna-se uma entrada mais curta. Na Figura 4-6(d), três entradas são fundidas, e dois itens são removidos da lista. Uma vez que a entrada na tabela de processos referente ao processo que está terminado normalmente apontará para a entrada de lista do próprio processo, pode ser mais conveniente ter a lista como uma lista duplamente encadeada, em vez de uma lista simplesmente encadeada, como a da Figura 4-5(c) Essa estrutura torna mais fácil localizar a entrada anterior e ver se uma fusão é possível.

Quando os processos e as lacunas são mantidos em uma lista classificada por endereço, vários algoritmos podem ser utilizados para alocar memória para um processo recentemente criado ou trocado para a memória. Supomos que o gerenciador de memória conhece a quantidade de memória a alocar. O algoritmo mais simples é chamado algoritmo do **primeiro ajuste**. O gerenciador de memória varre toda a lista de segmentos até localizar uma lacuna que seja suficientemente grande. A lacuna, então, é dividida em dois pedaços, um para o processo e um para a memória não-utilizada, exceto no improvável caso de um ajuste exato. O algoritmo do primeiro ajuste é rápido porque pesquisa o mínimo possível

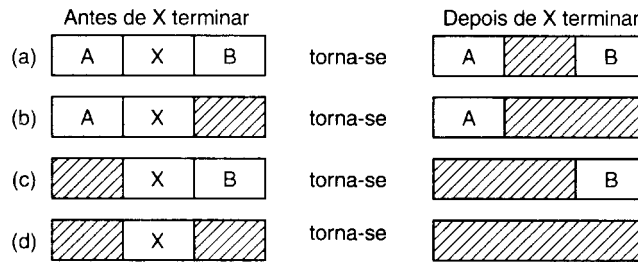



Figura 4-6 Quatro combinações de vizinho para o processo terminante *X*.

Uma variação secundária do algoritmo do primeiro ajuste é o do **próximo ajuste**. Ele funciona da mesma maneira que o primeiro ajuste, exceto que monitora a posição em que ele está sempre que encontra uma lacuna adequada. Da próxima vez que é chamado para localizar uma lacuna, ele começa pesquisando na lista a partir do lugar que ele deixou da última vez, em vez de sempre a partir do começo, como o primeiro ajuste faz. Simulações feitas por Bays (1977) demonstraram que o algoritmo do próximo ajuste oferece um desempenho ligeiramente inferior ao desempenho do algoritmo do primeiro ajuste.

Outro algoritmo bem-conhecido é o do **melhor ajuste**, o qual pesquisa na lista inteira e pega a menor lacuna que seja adequada. Antes de dividir uma lacuna grande que talvez seja necessária mais tarde, o melhor ajuste tenta localizar uma lacuna que é próxima do tamanho real necessário.

Como um exemplo do primeiro ajuste e do melhor ajuste, considere a Figura 4-5 novamente. Se um bloco de tamanho 2 for necessário, o primeiro ajuste alocará a lacuna em 5, mas o melhor ajuste alocará a lacuna em 18.

O melhor ajuste é mais lento que o primeiro ajuste porque deve pesquisar na lista inteira toda vez que é chamado. Um pouco surpreendentemente, ele também resulta em maior desperdício de memória que o primeiro ajuste ou o próximo ajuste porque tende a encher a memória de lacunas minúsculas e inúteis. O algoritmo de primeiro ajuste gera lacunas maiores, em média.

Para evitar o problema de dividir lacunas quase exatas entre um processo e uma lacuna minúscula pode-se pensar no **pior ajuste**, isto é, sempre pegar a maior lacuna disponível, de modo que a lacuna resultante seja suficientemente grande para ser útil. Contudo, simulações demonstraram que o pior ajuste também não é uma idéia muito boa.

Todos os quatro algoritmos podem ser acelerados mantendo-se listas separadas para processos e para lacunas. Assim, todos eles dedicam toda sua energia para localizar lacunas, não processos. O preço inevitável a ser pago por esse aceleramento da alocação é a complexidade adicional e a queda de velocidade ao se desalocar memória, uma vez que um segmento liberado tem que ser removido da lista de processos e inserido na lista de lacunas.

Se listas distintas são mantidas para processos e para lacunas, a lista de lacunas pode ser classificada por tamanho, tornando o algoritmo do melhor ajuste mais rápido. Quando o algoritmo de melhor ajuste pesquisa uma lista de lacunas do menor para o maior, logo que ele encontra uma lacuna que se ajusta, ele sabe que a lacuna é a menor que servirá para o *job*, daí o melhor ajuste. Nenhuma pesquisa adicional é requerida, como é necessário com o esquema de lista única. Com uma lista de lacunas classificada por tamanho, primeiro ajuste e melhor ajuste são igualmente rápidos, e o próximo ajuste é sem sentido.

Quando as lacunas são mantidas em listas separadas dos processos, é possível uma pequena otimização. Em vez de ter um conjunto separado de estruturas de dados para manter a lista de lacunas, como é feito na Figura 4-5(c), as próprias lacunas podem ser utilizadas. A primeira palavra de cada lacuna poderia ser o tamanho da lacuna, e a segunda palavra um ponteiro para a entrada seguinte. Os nós da lista da Figura 4-5(c), que requerem três palavras e um bit (P/H), não são mais necessários.

Ainda um outro algoritmo de alocação é o **ajuste rápido**, que mantém listas separadas para alguns dos tamanhos exigidos mais comuns. Por exemplo, você poderia ter uma tabela com *n* entradas, em que a primeira entrada é um ponteiro para a cabeça de uma lista de lacunas de 4K, a segunda entrada é um ponteiro para uma lista de lacunas de 8K, a terceira entrada um ponteiro para lacunas de 12K e assim por diante. As lacunas de, digamos, 21K, poderiam ser colocadas na lista de 20K ou em uma lista especial de lacunas de tamanhos variáveis. Com ajuste rápido, localizar uma lacuna do tamanho exigido é extremamente rápido, mas tem a mesma desvantagem que todos os esquemas que classificam por tamanho de lacuna, a saber, quando um processo termina ou está sendo enviado para disco, pesquisar seus vizinhos para ver se uma mescla é possível é caro. Se a mesclagem não for feita, a memória rapidamente será fragmentada em um grande número de pequenas lacunas em que nenhum processo ajusta-se.

4.3 MEMÓRIA VIRTUAL

Há muitos anos as pessoas defrontaram-se pela primeira vez com o problema dos programas que eram muito grandes para ajustar-se na memória disponível. A solução normalmente adotada era dividir o programa em pedaços,

chamados ***overlays***. O *overlay* 0 começava executando primeiro. Quando pronto, chamava outro *overlay*. Alguns sistemas de *overlay* eram altamente complexos, permitindo múltiplos *overlays* na memória simultaneamente. Os *overlays* eram mantidos em disco e eram levados para dentro e para fora da memória pelo sistema operacional, dinamicamente, conforme necessário.

Embora o trabalho real de comutação de *overlays* fosse feito pelo sistema, o trabalho de dividir o programa em pedaços tinha de ser feito pelo programador. Dividir programas grandes em pedaços pequenos e modulares exigia tempo e paciência. Não demorou muito para alguém pensar em uma maneira de atribuir todo o trabalho ao computador.

O método inventado (Fotheringham, 1961) veio a ser conhecido como **memória virtual.** A idéia básica por trás da memória virtual é que o tamanho combinado do programa, dos dados e da pilha pode exceder a quantidade de memória física disponível para ele. O sistema operacional mantém essas partes do programa atualmente em uso na memória principal e o restante em disco. Por exemplo, um programa de 16 M pode executar em uma máquina de 4 M, escolhendo cuidadosamente quais 4 M manter na memória a cada instante, com pedaços do programa sendo trocados entre o disco e a memória, conforme necessário.

A memória virtual também pode trabalhar em um sistema de multiprogramação, com pedaços de muitos programas na memória simultaneamente. Enquanto um programa está esperando parte dele próprio ser trazida para a memória, ele fica esperando a E/S e não pode executar; então, a CPU pode ser dada a outro processo, assim como em qualquer outro sistema de multiprogramação.

4.3.1 Paginação

Muitos sistemas de memória virtual utilizam uma técnica chamada **paginação**, que agora descreveremos. Em qualquer computador, existe um conjunto de endereços de memória que os programas podem produzir. Quando um programa utiliza uma instrução como

```
MOVE REG, 1000
```

ele está copiando o conteúdo do endereço de memória 1000 para REG (ou vice-versa, dependendo do computador). Os endereços podem ser gerados utilizando indexação, registradores de base, registradores de segmento e outras maneiras.

Esses endereços gerados por programa são chamados **endereços virtuais** e formam o **espaço de endereço virtual**; em computadores sem memória virtual, o endereço virtual é colocado diretamente no barramento da memória e faz com que a palavra física de memória com o mesmo endereço seja lida ou gravada. Quando é utilizada memória virtual, os endereços virtuais não vão diretamente para o barramento de memória. Em vez disso, eles vão para uma Unidade de Gerenciamento de Memória (**Memory Management Unit** — MMU), um chip ou uma coleção de chips que mapeia os endereços virtuais para os endereços físicos de memória como ilustrado na Figura 4-7.

Um exemplo muito simples de como esse mapeamento funciona é mostrado na Figura 4-8, no qual, temos um computador que pode gerar endereços de 16 bits, de 0 a 64K. Esses são os endereços virtuais. Esse computador, entretanto, tem somente 32K de memória física; então, embora programas de 64K possam ser escritos, eles não podem ser carregados na memória em sua totalidade e executados. Entretanto, uma cópia completa de uma imagem de núcleo do programa, até 64K, deve estar presente no disco, de modo que pedaços possam ser trazidos conforme necessário.

O espaço de endereço virtual é dividido em unidades chamadas **páginas**. As unidades correspondentes na memória física são chamadas **molduras de página**. As páginas e as molduras de página têm sempre exatamente o mesmo tamanho. Nesse exemplo, elas têm 4K, mas tamanhos de página de 512 bytes a 64K são comumente utilizados nos sistemas existentes. Com 64K de espaço de endere-

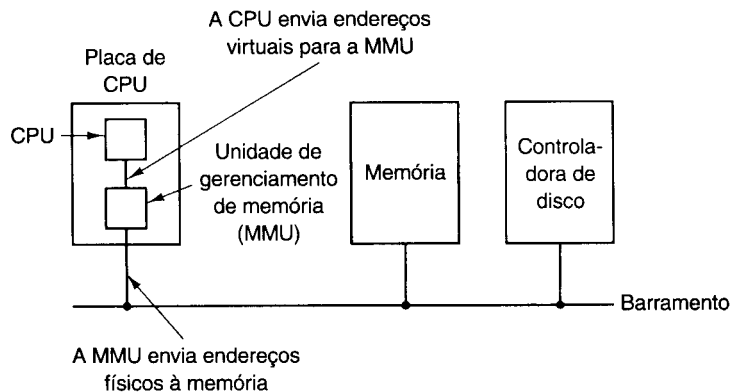



Figura 4-7 A posição e a função da MMU.

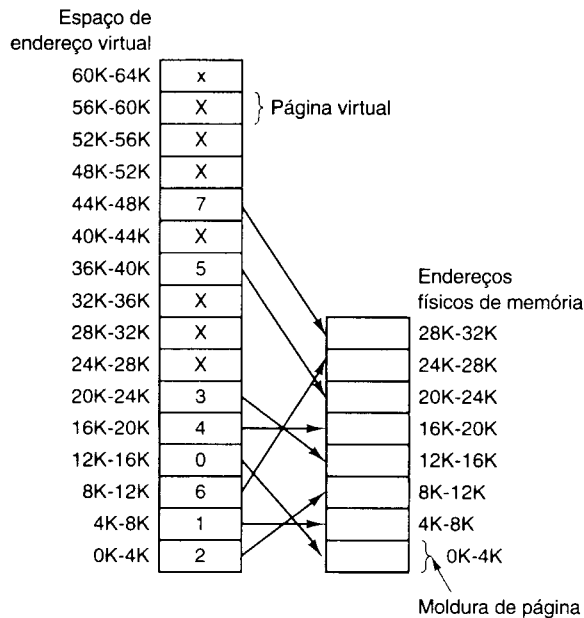



Figura 4-8 A relação entre endereços virtuais e endereços físicos de memória é dada pela tabela de páginas.

ço virtual e 32K de memória física, temos 16 páginas virtuais e 8 molduras de página. As transferências entre memória e disco são sempre em unidades de uma página.

Quando o programa tenta acessar o endereço 0, por exemplo, utilizando a instrução

MOVE REG,0

o endereço virtual 0 é enviado para a MMU. A MMU vê que esse endereço virtual cai na página 0 (0 a 4095), que, de acordo com seu mapeamento, é a moldura de página 2 (8192 a 12287). Assim, transforma o endereço para 8192 e coloca o endereço 8192 no barramento. A placa de memória não sabe absolutamente nada sobre a MMU e somente vê uma solicitação para ler ou para gravar o endereço 8192, que ela honra. Assim, a MMU efetivamente mapeou todos os endereços virtuais entre 0 e 4095 para os endereços físicos 8190 a 12287.

De maneira semelhante, uma instrução

MOVE REG,8192

é efetivamente transformada em

MOVE REG,24576

porque endereço virtual 8192 está na página virtual 2 e essa página é mapeada para a moldura de página física 6 (endereços físicos 24576 até 28671). Como um terceiro exemplo, o endereço virtual 20500 está a 20 bytes do início da página virtual 5 (endereços virtuais 20480 a 24575) e é mapeado sobre endereço físico 12288 + 20 = 12308.

Por si mesmo, essa capacidade de mapear as 16 páginas virtuais para qualquer um das 8 molduras de página, configurando o mapa da MMU apropriadamente não resolve o problema de que o espaço de endereço virtual é maior que a memória física. Uma vez que fisicamente temos apenas 8 molduras de página, somente 8 das páginas virtuais na Figura 4-8 são mapeadas na memória física. Os outros, mostrados como um X na figura, não são mapeados. No hardware real, um bit **presente/ausente** em cada entrada monitora se a página é mapeada ou não.

O que acontece se o programa tenta utilizar uma página não-mapeada, por exemplo, utilizando a instrução

MOVE REG,32780

que é o byte 12 dentro da página virtual 8 (iniciando em 32768)? A MMU nota que a página não está mapeada (indicada por um xis na figura) e gera uma interrupção, passando a CPU para o sistema operacional. Tal interrupção é chamada **falha de página.** O sistema operacional seleciona uma moldura de página pouco utilizada e grava o seu conteúdo de volta no disco. Então, ele busca a página que acabou de ser referenciada e carrega-a na moldura de página que acabou de ser liberada, altera o mapa e reinicia a instrução interrompida.

Por exemplo, se o sistema operacional decidisse liberar a moldura de página 1, ele carregaria a página virtual 8 no endereço físico 4K e faria duas alterações no mapa da MMU. Primeiro, marcaria a entrada da página virtual 1 como não-mapeada, para impedir qualquer futuro acesso a endereços virtuais entre 4K e 8K. Então, substituiria o X na entrada de página virtual 8 por um 1, de modo que quando a instrução interrompida é reexecutada, ela mapeará o endereço virtual 32780 para o endereço físico 4108.

Agora vamos olhar dentro da MMU para ver como ela funciona e por que escolhemos utilizar um tamanho de página que é uma potência de 2. Na Figura 4-9, vemos um exemplo de um endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado utilizando o mapa de MMU da Figura 4-8. O endereço virtual de 16 bits que chega é dividido em um número de página de 4 bits e um deslocamento de 12 bits. Com 4 bits para o número de página, podemos representar 16 páginas; e com 12 bits para o deslocamento, podemos endereçar todos os 4096 bytes dentro de uma página.

O número de página é utilizado como um índice na **tabela de páginas**, que fornece o número da moldura de página correspondente a essa página virtual. Se o bit *presente/ausente* for 0, uma interrupção é gerada, passando a CPU para o sistema operacional. Se o bit for 1, o número da moldura de página localizado na tabela de páginas será copiado para os 3 bits de ordem superior do registrador de saída, junto com o deslocamento de 12 bits, que é copiado sem modificações do endereço virtual dado. Juntos, eles formam um endereço físico de 15 bits. O registrador de saída, então, é colocado sobre o barramento de memória como o endereço físico de memória.

4.3.2 Tabelas de Página

Na teoria, o mapeamento de endereços virtuais para endereços físicos é como acabamos de descrever. O endereço virtual é dividido em um número de página virtual (bits de ordem superior) e um deslocamento (bits de ordem inferior). O número de página virtual é utilizado como um índice na tabela de páginas para localizar a entrada para essa página virtual. A partir da entrada da tabela de páginas, o número da moldura de página (se há alguma) é localizado. O número da moldura de página é anexado à extremidade de ordem superior do deslocamento, substituindo o número de página virtual, para formar um endereço físico que pode ser enviado para a memória.

O propósito da tabela de páginas é mapear páginas virtuais em molduras de página. Matematicamente falando, a tabela de páginas é uma função, com o número de página virtual como argumento, e o número da moldura física como resultado. Utilizando o resultado dessa função, o campo de página virtual em um endereço virtual pode ser substituído por um campo de moldura de página, formando, assim, um endereço físico de memória.

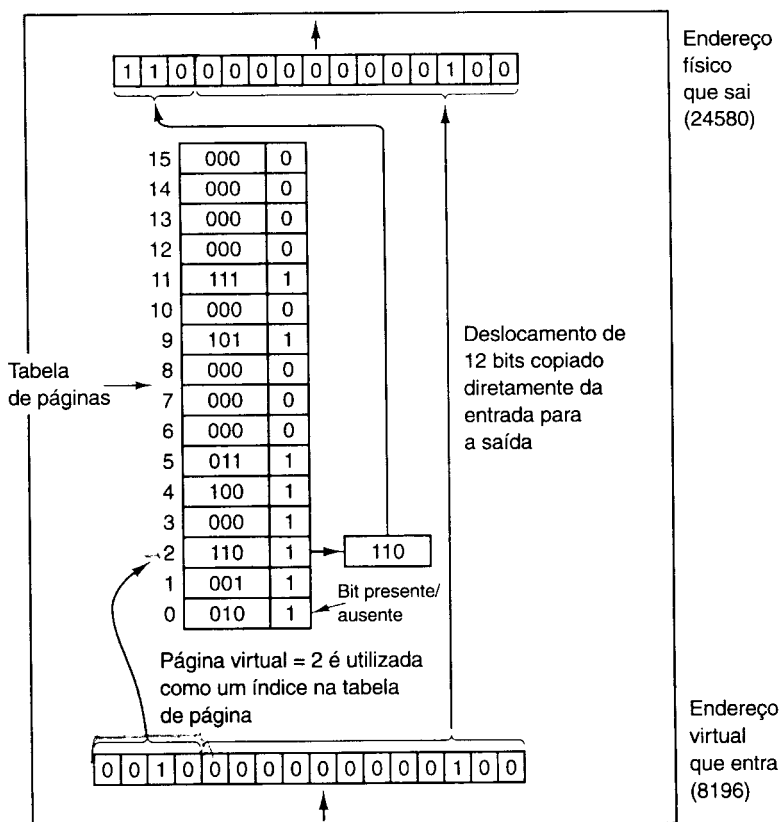



Figura 4-9 A operação interna da MMU com 16 páginas 4K.

Apesar dessa descrição simples, duas questões importantes devem ser encaradas:

1. A tabela de páginas pode ser extremamente grande.
2. O mapeamento deve ser rápido.

O primeiro ponto é conseqüência do fato de que computadores modernos utilizam endereços virtuais de pelo menos 32 bits. Com, digamos, um tamanho de página de 4K, um espaço de endereçamento de 32 bits de endereço tem 1 milhão de páginas, e um espaço de endereçamento de 64 bits tem mais do que você pode contemplar. Com 1 milhão de páginas, no espaço de endereçamento virtual, a tabela de páginas deve ter 1 milhão de entradas. E lembre-se de que cada processo necessita da própria tabela de páginas.

O segundo ponto é uma conseqüência do fato de que o mapeamento de virtual para físico deve ser feito em cada referência da memória. Uma instrução típica tem uma palavra de instrução e freqüentemente um operando de memória também. Portanto, é necessário fazer 1, 2 ou, às vezes, mais referências da tabela de páginas por instrução. Se uma instrução toma, digamos, 10ns, a pesquisa na tabela de páginas deve ser feita em poucos nanossegundos para que não se torne um gargalo importante.

A necessidade de um mapeamento de páginas rápido e grande é uma limitação significativa na maneira como os computadores são construídos. Embora o problema mais sério seja com máquinas topo de linha, é também uma questão na extremidade baixa, onde custo e preço/desempenho são críticos. Nesta seção e nas seguintes, veremos o projeto da tabela de páginas detalhadamente e mostraremos algumas soluções de hardware que foram utilizadas em computadores reais.

O projeto mais simples (pelo menos conceitualmente) é ter uma única tabela de páginas consistindo em uma matriz de rápidos registradores de hardware, com uma entrada para cada página virtual, indexada pelo número de página virtual. Quando um processo é iniciado, o sistema operacional carrega os registradores com a tabela de páginas do processo, tomada de uma cópia mantida na memória principal. Durante a execução do processo, referências de memória não são mais requeridas para a tabela de página. As vantagens desse método são que ele é simples e direto e não exige nenhuma referência de memória durante o mapeamento. Uma desvantagem é que é potencialmente caro (se a tabela de páginas é grande). Ter de carregar a tabela de páginas em cada comutação de contexto também pode prejudicar o desempenho.

No outro extremo, a tabela de páginas pode estar inteiramente na memória principal. Tudo o que o hardware precisa, então, é de um único registrador que aponta para o início da tabela de páginas. Esse projeto permite que o mapa de memória seja alterado em uma comutação de contexto, recarregando um registrador. Naturalmente, tem a desvantagem de exigir que uma ou mais referências de memória leiam entradas da tabela de páginas durante a execução de cada instrução. Por essa razão, essa abordagem raramente é utilizada em sua forma mais pura, mas a seguir estudaremos algumas variações que têm desempenho muito melhor.

Tabelas de Páginas Multinível

Para evitar o problema de ter enormes tabelas de página na memória todo o tempo, muitos computadores utilizam uma tabela de páginas multinível. Um exemplo simples é mostrado na Figura 4-10. Na Figura 4-10(a), temos um endereço virtual de 32 bits que é particionado em um campo *PT1* de 10 bits, um campo *PT2* de 10 bits e um campo de deslocamento de 12 bits. Uma vez que os deslocamentos são de 12 bits, as páginas são de 4K e há um total de 2^{20} deles.

O segredo para o método de tabela de páginas multinível é evitar manter todas as tabelas de página na memória todo o tempo. Em particular, aquelas que não são necessárias não devem ser mantidas. Suponha, por exemplo, que um processo necessite de 12 megabytes, os 4 megabytes da parte inferior da memória para texto do programa, os próximos 4 megabytes para dados e os 4 megabytes da parte superior para a pilha. Entre a porção acima dos dados e a porção baixa das pilha há uma lacuna gigantesca que não é utilizada.

Na Figura 4-10(b) vemos como a tabela de páginas de dois níveis funciona nesse exemplo. Na esquerda, temos a tabela de páginas de primeiro nível, com 1024 entradas, correspondente ao campo de 10 bits *PT1*. Quando um endereço virtual é apresentado para a MMU, ela primeiro extrai o campo *PT1* e utiliza esse valor como um índice na tabela de páginas de primeiro nível. Cada uma dessas 1024 entradas representa 4 M porque o espaço de endereçamento virtual de 4 gigabytes inteiro (i. e., de 32 bits) foi dividido em pedaços de 1024 bytes.

A entrada localizada na pesquisa na tabela de páginas de primeiro nível fornece o endereço ou o número da moldura de página de uma tabela de páginas de segundo nível. A entrada 0 da tabela de páginas de primeiro nível aponta para a tabela de páginas do texto do programa, a entrada 1 aponta para a tabela de páginas dos dados e a entrada 1023 aponta para a tabela de páginas da pilha. As outras entradas (sombreadas) não são utilizadas. O campo *PT2* é agora utilizado como um índice na tabela de páginas de segundo nível selecionada para localizar o número da moldura de página para a página em si.

Como um exemplo, considere o endereço virual de 32 bits 0x00403004 (4.206.596 em decimal), que está 12.292 bytes nos dados. Esse endereço corresponde a *PT1* = *1*, *PT2* = 3 e deslocamento = 4. A MMU primeiro utiliza *PT1* para indexar na tabela de páginas de primeiro nível e para obter a entrada 1, que corresponde aos endereços de 4 M a 8 M. Então, ela utiliza *PT2* para indexar na tabela de páginas de segundo nível recém-localizada e extrair a entrada 3, que corresponde aos endereços de 12288 a 16383 dentro de seu bloco de 4 M (i. e., endereços absolutos de 4.206.592 até 4.210.687). Essa entrada contém o número da moldura de página da página onde se encontra o endereço virtu-

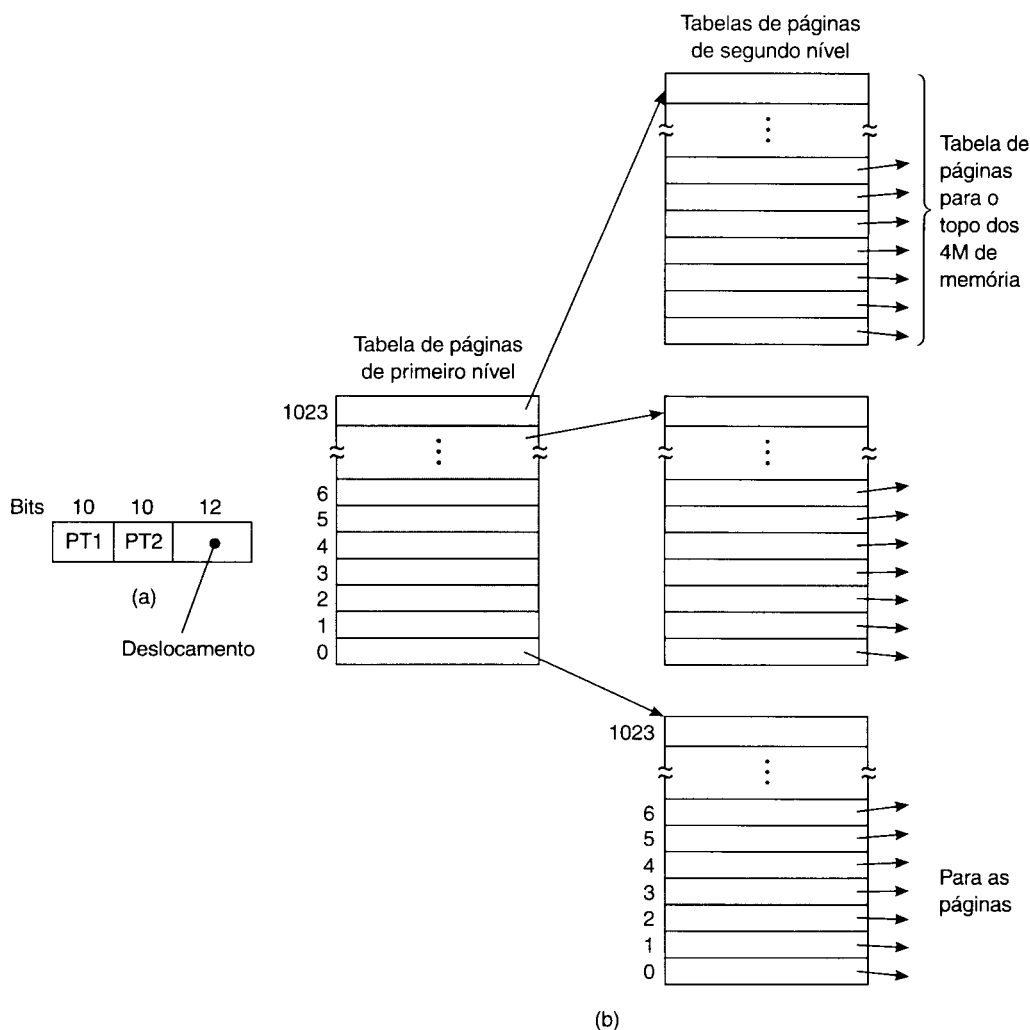



Figura 4-10 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.

al 0x00403004. Se essa página não estiver na memória, o bit *presente/ausente* na entrada da tabela de páginas será zero, causando uma falha de página. Se a página estiver na memória, o número da moldura de página tomado da tabela de páginas de segundo nível é combinado com o deslocamento (4) para construir um endereço físico. Esse endereço é colocado no barramento e enviado para a memória.

A coisa interessante a observar sobre a Figura 4-10 é que embora o espaço de endereçamento contenha mais de um milhão de páginas, somente quatro tabelas de página realmente são exigidas: a tabela de primeiro nível e as de segundo nível para 0 a 4M, 4 M a 8 M e para os 4 M superiores. Os bits *presente/ausente* em 1021 entradas da tabela de páginas de primeiro nível são configurados como 0, forçando uma falha de página se elas forem acessadas. Se isso ocorrer, o sistema operacional receberá que o processo está tentando referenciar memória que não lhe foi destinada e tomará ação apropriada, como enviar a ele um sinal ou eliminá-lo. Nesse exemplo, escolhemos números redondos para os vários tamanhos e selecionamos *PT1* igual a *PT2*, mas, na prática, outros valores também são possíveis, naturalmente.

O sistema de tabela de páginas de dois níveis da Figura 4-10 pode ser expandido para três, para quatro ou para mais níveis. Níveis adicionais dão mais flexibilidade, mas é duvidoso que a complexidade adicional valha a pena além de três níveis.

Agora vamos deixar a estrutura das tabelas de páginas no geral e ir para os detalhes de uma única entrada de tabela de páginas. O arranjo exato de uma entrada é alta-

mente dependente de máquina, mas os tipos de informações presentes são grosseiramente os mesmos de máquina para máquina. Na Figura 4-11, damos uma entrada de tabela de página de exemplo. O tamanho varia de computador para computador, mas 32 bits é um tamanho comum. O campo mais importante é o número da moldura de página. Afinal de contas, o objetivo do mapeamento de página é localizar esse valor. Perto dele, temos o bit *presente/ausente*. Se esse bit for 1, a entrada é válida e pode ser utilizada. Se for 0, a página virtual a que a entrada pertence não está atualmente na memória. Acessar uma entrada da tabela de páginas com esse bit configurado como zero causa uma falha de página.

Os bits *proteção* informam que tipos de acesso são permitidos. Na forma simples, esse campo contém 1 bit, com 0 para leitura/gravação e 1 para leitura somente. Um arranjo mais sofisticado é ter 3 bits, cada um para habilitar leitura, gravação e execução da página.

Os bits *modificada* e *referenciada* monitoram o uso da página. Quando se escreve em uma página, o hardware automaticamente liga o bit *modificada*. Esse bit é valioso quando o sistema operacional decide reivindicar uma moldura de página. Se a página nela foi modificada (i. e., está "suja"), ela deve ser gravada de volta no disco. Se ela não foi modificada (i. e., está "limpa"), ela pode ser simplesmente abandonada, uma vez que a cópia no disco ainda é válida. O bit às vezes é chamado de **bit sujo**, uma vez que reflete o estado da página.

O bit *referenciada* é ligado sempre que uma página é referenciada, seja para ler ou para gravar. Seu objetivo é ajudar o sistema operacional a escolher uma página a expulsar quando uma falha de página ocorre. As páginas que não estão sendo utilizadas são melhores candidatas que as páginas que estão, e esse bit desempenha um papel importante em vários dos algoritmo de substituição de página que veremos mais adiante neste capítulo.

Por fim, o último bit permite que o *cache* seja desativado para a página. Esse recurso é importante para páginas que são mapeadas em registradores de dispositivo em vez da memória. Se o sistema operacional está preso em um laço estrito, esperando algum dispositivo de E/S responder a um comando que acaba de dar, é essencial que o hardware continue buscando a palavra do dispositivo e não utilize uma cópia de *cache* antiga. Com esse bit, a operação de *cache* pode ser desligada. As máquinas que têm um espaço separado de E/S e não utilizam E/S mapeada em memória não necessitam desse bit.

Note que o endereço de disco utilizado para armazenar a página quando não está na memória não é parte da tabela de páginas. A razão é simples. A tabela de página armazena somente as informações de que o hardware precisa para traduzir um endereço virtual em um endereço físico. As informações que o sistema operacional necessita para tratar falhas de página são mantidas em tabelas de software dentro do sistema operacional.

4.3.3 TLBs — *Translation Lookaside Buffers*

Na maioria dos esquemas de paginação, as tabelas de página são mantidas na memória, devido ao seu tamanho grande. Potencialmente, esse projeto tem um enorme impacto sobre o desempenho. Considere, por exemplo, uma instrução que copia um registrador para outro. Na ausência de paginação, essa instrução faz somente uma referência à memória, para buscar a instrução. Com a paginação, referências de memória adicionais serão necessárias para acessar a tabela de páginas. Uma vez que a velocidade de execução geralmente é limitada pela taxa com que a CPU pode obter instruções e dados da memória, ter de fazer duas referências de tabela de páginas por referência de memória reduz o desempenho em 2/3. Sob essas condições, ninguém a utilizaria.

Os projetistas de computador sabiam desse problema havia anos e ofereceram uma solução. Sua solução é baseada na observação de que a maioria dos programas tende a fazer um grande número de referências a um pequeno número de páginas, e não o contrário. Assim somente uma pequena fração de entradas da tabela de páginas é intensamente lida; o restante é muito pouco utilizado.

A solução inventada foi equipar computadores com um pequeno dispositivo de hardware para mapear endereços virtuais em endereços físicos sem passar pela tabela de páginas. O dispositivo, chamado **TLB** (***Translation Lookaside Buffer***) ou, às vezes, **memória associativa**, é ilustrado na Figura 4-12. Normalmente, ele está dentro da MMU e consiste em um pequeno número de entradas, oito nesse exemplo, mas raramente mais de 64. Cada entrada con-

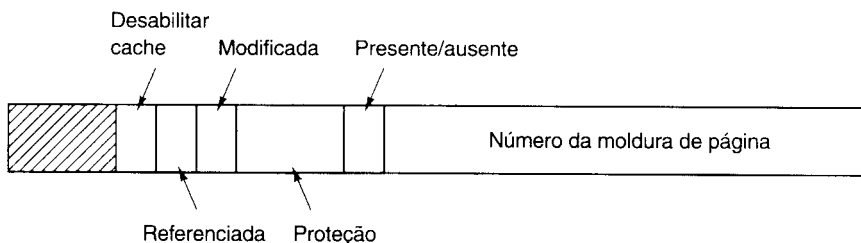



Figura 4-11 Uma entrada típica de tabela de páginas.

têm as informações sobre uma página, em particular, o número de página virtual, um bit que é ligado quando a página é modificada, o código de proteção (permissões para ler/gravar/executar) e a moldura de página física em que a página está localizada. Esses campos têm correspondência um para um com os campos na tabela de páginas. Outro bit indica se a entrada é válida (i. e., está em uso) ou não.

Um exemplo que pode gerar o TLB da Figura 4-12 é um processo em um laço que se estende pelas páginas virtuais 19, 20 e 21, de tal modo que essas entradas de TLB têm códigos de proteção para ler e executar. Os dados principais atualmente sendo utilizados (digamos, uma matriz que está sendo processada) estão nas páginas 129 e 130. A página 140 contém os índices utilizados nos cálculos da matriz. Por fim, a pilha está nas páginas 860 e 861.

Vejamos agora como o TLB funciona. Quando um endereço virtual é apresentado para a MMU para tradução, o hardware primeiro verifica se seu número de página virtual está presente no TLB, comparando-o com todas as entradas simultaneamente (i. e., em paralelo). Se uma coincidência válida for localizada, e o acesso não violar os bits de proteção, a moldura de página será tomada diretamente do TLB, sem passar pela tabela de páginas. Se o número de página virtual estiver presente no TLB, mas a instrução estiver tentando gravar em uma página com permissão apenas de leitura, uma falha de proteção será gerada, da mesma maneira como seria a partir da própria tabela de páginas.

O caso interessante é o que acontece quando o número de página virtual não está no TLB. A MMU detecta a falta e faz uma pesquisa rotineira na tabela de páginas. Então, ela expulsa uma das entradas do TLB e substitui-a pela entrada da tabela de páginas recém-pesquisada. Assim, se essa página for utilizada novamente logo, a segunda vez resultará em um acerto em vez de uma falta. Quando uma entrada é purgada do TLB, o bit *modificada* é copiado de volta para a entrada da tabela de páginas na memória. Os outros valores já estão aí. Quando o TLB é carregado da tabela de páginas, todos os campos são tomados da memória.

Gerenciamento por Software do TLB

Até agora, assumimos que cada máquina com memória virtual paginada tem tabelas de páginas reconhecidas pelo hardware, mais um TLB. Nesse projeto, gerenciamento de TLB e tratamento de falhas de TLB são feitos inteiramente pelo hardware de MMU. As interrupções para o sistema operacional ocorrem somente quando uma página não está na memória.

No passado, essa suposição era verdadeira. Entretanto, algumas máquinas RISC modernas, incluindo MIPS, Alfa e HP PA, fazem quase todo o gerenciamento de páginas em software. Nessas máquinas, as entradas de TLB são explicitamente carregadas pelo sistema operacional. Quando uma falta de TLB ocorre, em vez de a MMU simplesmente ir para a tabela de páginas localizar e buscar a referência de página necessária, ela apenas gera uma falha de TLB e joga o problema para o sistema operacional. O sistema deve localizar a página, remover uma entrada do TLB, entrar de novo e reiniciar a instrução que falhou. E, naturalmente, tudo isso deve ser feito em um punhado de instruções porque faltas de TLB ocorrem com muito mais freqüência que falhas de paginação.

Surpreendentemente, se o TLB for razoavelmente grande (digamos, 64 entradas) para reduzir a taxa de falta, o gerenciamento por software do TLB vem a ser bastante eficiente. O ganho principal aqui é uma MMU muito mais simples, o que libera uma quantidade considerável de área no chip da CPU para *caches* e para outros recursos que podem melhorar o desempenho. O gerenciamento por software do TLB é discutido demoradamente por Uhlig e colaboradores (1994).

Várias estratégias foram desenvolvidas para melhorar o desempenho em máquinas que fazem gerenciamento do TLB por software. Uma abordagem é reduzir as faltas de TLB e reduzir os custos de uma falta de TLB quando ocorre (Bala *et al.*, 1994). Para reduzir faltas de TLB, às vezes, o sistema operacional pode utilizar sua intuição para descobrir que páginas podem ser utilizadas em seguida e pré-carregar entradas para elas no TLB. Por exemplo, quando um processo de cliente faz uma RPC* para um processo de servidor na mesma máquina, é muito possível que o servidor precisará executar em breve. Sabendo disso, ao processar a interrupção para fazer o RPC, o sistema também pode verificar onde o código do servidor, os dados e as páginas de pilha estão e mapeá-los antes de eles poderem causar falhas de TLB.

A maneira normal de processar uma falta de TLB, em hardware ou em software, é ir para a tabela de páginas e executar as operações de indexação para localizar a página referenciada. O problema ao fazer essa pesquisa em software é que as páginas que armazenam a tabela de páginas podem não estar no TLB, o que causará falhas adicionais de TLB durante o processamento. Essas falhas podem ser reduzidas mantendo um grande (p. ex., 4K) *cache* de software de entradas de TLB em uma posição fixa cuja pagina sempre é mantida no TLB. Por primeiro verificar o *cache* de software, o sistema operacional pode reduzir substancialmente faltas de TLB.

4.3.4 Tabelas de Páginas Invertidas

Tabelas de páginas tradicionais, do tipo descrito até agora, exigem uma entrada por página virtual, uma vez que são indexadas por número de página virtual. Se o espaço de endereçamento consiste em 2^{32} bytes, com 4096 bytes por página, então, são necessárias mais de 1 milhão de entradas de tabela de página. Como um mínimo, a ta-

*N. de T. *Remote Procedure Call*, chamada remota de procedimento.

Válida	Página virtual	Modificada	Proteção	Moldura de página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 4-12 Um TLB para acelerar a paginação.

bela de páginas deverá ter pelo menos 4 megabytes. Em sistemas maiores, esse tamanho é provavelmente factível.

Entretanto, à medida que computadores de 64 bits tornam-se mais comuns, a situação muda drasticamente. Se o espaço de endereço é agora de 2^{64} bytes com páginas de 4K, necessitamos mais de 10^{15} bytes para a tabela de páginas. Prender mais de 1 milhão de gigabytes somente para a tabela de páginas não é aceitável, não por enquanto e não pelas próximas décadas, se é que algum dia será. Portanto, uma solução diferente é requerida para espaços de endereço virtual paginados de 64 bits.

A solução para isso é a **tabela de páginas invertida.** Nesse projeto, há uma entrada por moldura de página na memória real, em vez de uma entrada por página de espaço de endereço virtual. Por exemplo, com endereços virtuais de 64 bits, uma página 4K e 32MB de RAM, uma tabela de página invertida somente precisa de 8192 entradas. A entrada monitora qual (processo, página virtual) está localizado na moldura de página.

Embora as tabelas de página invertidas economizem quantidades significativas de espaço, pelo menos quando o espaço de endereço virtual é muito maior que a memória física, elas têm uma desvantagem séria: a tradução de virtual para físico torna-se muito mais difícil. Quando o processo n referencia a página virtual p, o hardware não pode mais localizar a página física utilizando p como um índice na tabela de página. Em vez disso, ele deve pesquisar na tabela de página invertida inteira uma entrada (n,p). Além disso, essa pesquisa deve ser feita em cada referência de memória, não somente em falhas de página. Pesquisar uma tabela de 8K em cada referência de memória não é a maneira certa de tornar rápida sua máquina.

A saída para esse dilema é utilizar o TLB. Se o TLB pode armazenar todas as páginas mais utilizadas, a tradução pode acontecer igualmente rápida como com tabelas de páginas comuns. Em uma falta de TLB, entretanto, a tabela de página invertida precisa ser pesquisada. Entretanto, utilizando uma tabela de *hash* como um índice na tabela de página invertida, essa pesquisa pode ser tornada razoavelmente rápida. As tabelas de página invertidas são atualmente utilizadas em algumas estações de trabalho IBM e Hewlett-Packard e vão se tornar mais comuns à medida que as máquinas de 64 bits difundirem-se.

Outra abordagem para tratamento de memórias virtuais grandes pode ser encontrada em (Huck e Hays, 1993; Talluri e Colina, 1994: e Talluri *et al.*, 1995).

4.4 ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINA

Quando ocorre uma falha de página, o sistema operacional precisa escolher uma página para remover da memória e dar lugar para a página que precisa ser carregada. Se a página a ser removida tiver sido modificada enquanto na memória, ela deve ser regravada no disco para atualizar a cópia em disco. Se, entretanto, a página não foi alterada (p. ex., uma página contém texto de programa), a cópia em disco já está atualizada; então, nenhuma regravação é exigida. A página a ser lida simplesmente sobrescreve a página sendo expulsa.

Embora seja possível selecionar aleatoriamente uma página para remover em cada falha de página, o desempenho do sistema será muito melhor se uma página que não é intensamente utilizada for escolhida. Se uma página intensamente utilizada for removida, provavelmente ela precisará ser trazida de volta em breve, o que resulta em sobrecarga extra. Muitos trabalhos foram feitos sobre o assunto de algoritmo de substituição de página, tanto teóricos como experimentais. A seguir, descreveremos alguns dos algoritmos mais importantes.

4.4.1 Algoritmo de Substituição de Página Ótimo

O melhor algoritmo de substituição de página possível é fácil de descrever mas impossível de implementar. Acompanhe o raciocínio. No momento em que uma falha de página ocorre, algum conjunto de páginas está na memória. Uma dessas páginas será referenciada na instrução mais

próxima (a página que contém essa instrução). Outras páginas podem não ser referenciadas até 10, 100 ou talvez 1.000 instruções mais tarde. Cada página pode ser rotulada com o número de instruções que será executado antes de essa página ser referenciada pela primeira vez.

O algoritmo de página ótimo simplesmente diz que a página com o rótulo mais alto deve ser removida. Se uma página não será utilizada por 8 milhões de instruções e outra página não será utilizada por 6 milhões de instruções, remover a primeira empurra a falha de página que a buscará de volta para o mais tarde possível. Os computadores, como as pessoas, tentam adiar eventos desagradáveis o máximo que puderem.

O único problema com esse algoritmo é que ele não é realizável. No momento da falha de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada em seguida. (Vimos uma situação semelhante anteriormente com o algoritmo de agendamento *job* mais curto primeiro — como o sistema pode dizer qual *job* é mais curto?) Ademais, executando um programa em um simulador e monitorando todas as referências de página, é possível implementar substituição de página ótima na *segunda* execução utilizando as informações de referência de página coletadas durante a *primeira* execução.

Dessa maneira é possível comparar o desempenho de algoritmos realizáveis com o melhor possível. Se um sistema operacional alcançar um desempenho de, digamos, somente 1% pior do que o algoritmo ótimo, o esforço gasto em pesquisar um melhor algoritmo resultará em uma melhora de 1% no máximo

Para evitar qualquer possível confusão, deve-se deixar claro que esse registro de referências de página refere-se apenas ao programa recém-medido. O algoritmo de substituição de página derivado dele é específico desse programa. Embora esse método seja útil para avaliar algoritmos de substituição de página, ele não serve para nada em sistemas práticos. A seguir, estudaremos algoritmos que *são* úteis em sistemas reais.

4.4.2 Algoritmo de Substituição de Página Não Recentemente Utilizada

Para permitir que o sistema operacional colecione estatísticas úteis sobre quais páginas estão sendo utilizadas e quais não estão, a maioria dos computadores com memória virtual tem dois bits de status associados a cada página. *R* é configurado sempre que a página é referenciada (leitura ou gravação). *M* é configurado quando a página é gravada (i. e., modificada). Os bits são contidos em cada entrada da tabela de páginas, como mostrado na Figura 4-11. É importante saber que esses bits devem ser atualizados em cada referência de memória, então, é essencial que sejam configurados pelo hardware. Uma vez que um bit foi configurado como 1, ele permanece como 1 até que o sistema operacional o redefina para 0 em software.

Se o hardware não tiver esses bits, eles podem ser simulados da seguinte maneira. Quando um processo é iniciado, todas as entradas da tabela de páginas são marcadas como não estando na memória. Logo que qualquer página é referenciada, ocorrerá uma falha de página. O sistema operacional, então, liga o bit *R* (em suas tabelas internas), altera a entrada da tabela de páginas para apontar para a página correta, com modo apenas para leitura, e reinicia a instrução. Se a página for subseqüentemente gravada, ocorrerá outra falha de página, permitindo que o sistema operacional ligue o bit *M* e altere o modo da página para leitura/escrita.

O bits *R* e *M* podem ser utilizados para construir um algoritmo de paginação simples como segue. Quando um processo é iniciado, os dois bits de página para todas as suas páginas são configurados como 0 pelo sistema operacional. Periodicamente (p. ex., a cada interrupção do relógio), o bit *R* é limpo, distinguindo páginas que não foram referenciadas recentemente das que foram.

Quando ocorre uma falha de página, o sistema operacional inspeciona todas as páginas e divide-as em quatro categorias com base nos valores atuais de seu bits *R* e *M*:

A classe 0: não-referenciada, não-modificada.
A classe 1: não-referenciada, modificada.
A classe 2: referenciada, não-modificada.
A classe 3: referenciada, modificada.

Embora as páginas de classe 1 pareçam impossíveis à primeira vista, elas ocorrem quando uma página de classe 3 tem seu bit *R* limpo por uma interrupção de relógio. A interrupção de relógio não limpa o bit *M* porque essa informação é necessária para saber se a página tem de ser regravada no disco ou não.

O algoritmo **NRU (Não Recentemente Utilizada)** remove uma página em execução da classe não-vazia com a numeração mais baixa. Implícito nesse algoritmo está o fato de que é melhor remover uma página modificada que não foi referenciada no período de um tique de relógio (em geral 20ms) do que uma página limpa que está em uso intenso. A principal atração do NRU é que é fácil de entender, sua implementação é eficiente e ele oferece um desempenho que, embora certamente não-ótimo, é freqüentemente adequado.

4.4.3 Algoritmo de Substituição de Página Primeira a Entrar, Primeira a Sair (FIFO)

Outro algoritmo de paginação de baixo *overhead* é o algoritmo **FIFO (*First-In, First-Out*)**. Para ilustrar como isso funciona, considere um supermercado que tem prateleiras suficientes para exibir exatamente *k* produtos diferentes. Um dia, uma empresa introduz um novo alimento

de conveniência — um tipo de iogurte congelado e seco para preparo instantâneo, que pode ser reconstituído em um forno de microondas. O produto é um sucesso imediato, então, nosso limitado supermercado precisa livrar-se de um produto antigo para armazenar o novo.

Uma possibilidade é localizar o produto que o supermercado armazena há mais tempo (i. e., algo que começou a vender 120 anos atrás) e livrar-se dele com base no fato de que ninguém está mais interessado nele. Por sorte, o supermercado mantém uma lista de todos os produtos atualmente vendidos na ordem em que eles foram introduzidos. O novo entra no fim na lista; aquele que está no começo da lista é derrubado.

Como um algoritmo de substituição de página, a mesma idéia é aplicável. O sistema operacional mantém uma lista de todas as páginas atualmente na memória, sendo que a página no topo da lista é a mais antiga, e a página no fim é a mais recente. Em uma falha de página, a página no topo é removida e a nova página é adicionada no fim da lista. Quando aplicado a estoques, FIFO talvez remova produtos como vaselina para bigodes, mas talvez também remova farinha, sal ou manteiga. Quando aplicado a computadores, o mesmo problema surge. Por essa razão, FIFO em sua forma pura é raramente utilizado.

4.4.4 Algoritmo de Substituição de Página de Segunda Chance

Uma modificação simples do FIFO que evita o problema de jogar fora uma página intensamente utilizada é inspecionar o bit R da página mais antiga. Se for 0, a página será antiga ou não-utilizada, então, ela é substituída imediatamente. Se o bit R for 1, o bit será limpo, a página será colocada no fim da lista de páginas e seu tempo de carga é atualizado como se acabasse de chegar na memória. Então, a pesquisa continua.

A operação desse algoritmo, chamado algoritmo de **segunda chance**, é mostrada na Figura 4-13. Na Figura 4-13(a), vemos páginas A a H mantidas em uma lista encadeadas e classificadas pelo tempo em que chegaram à memória.

Suponha que uma falha de página ocorra no tempo 20. A página mais antiga é A, que chegou no tempo 0, quando o processo iniciou. Se o bit R de A estiver limpo, ela é expulsa da memória, seja sendo gravada no disco (se estiver suja) ou simplesmente sendo abandonada (se estiver limpa). Por outro lado, se o bit R estiver ligado, A é colocada no fim da lista e seu "tempo de carga" é reconfigurado como o tempo atual (20). O bit R também é limpo. A busca por uma página adequada continua com B.

O que o algoritmo de segunda chance está fazendo é pesquisar uma página antiga que não foi referenciada no intervalo de relógio anterior. Se todas as páginas foram referenciadas, o algoritmo de segunda chance degenera em FIFO puro. Especificamente, imagine que todas as páginas na Figura 4-13(a) tenham seus bits R ligados. Um por um, o sistema operacional move as páginas para o fim da lista, e limpa o bit R cada vez que anexa uma página ao fim da lista. Por fim, ele volta para a página A, que agora tem seu bit R limpo. Nesse ponto, A é expulsa. Assim o algoritmo sempre termina.

4.4.5 Algoritmo de Substituição de Página do Relógio

Embora o algoritmo de segunda chance seja razoável, é desnecessariamente ineficiente porque constantemente está movendo páginas em sua lista. Uma abordagem melhor é manter todas as páginas em uma lista circular na forma de um relógio, como mostrado na Figura 4-14. Um ponteiro indica a página mais antiga.

Quando uma falha de página ocorre, a página indicada pelo ponteiro é inspecionada. Se seu bit R for 0, a página será expulsa, a nova página será inserida no relógio em seu lugar e o ponteiro será avançado uma posição. Se R for 1, ele será limpo, e o ponteiro é avançado para a próxima página. Esse processo é repetido até que uma página seja localizada com $R = 0$. Não é de surpreender que esse algo-

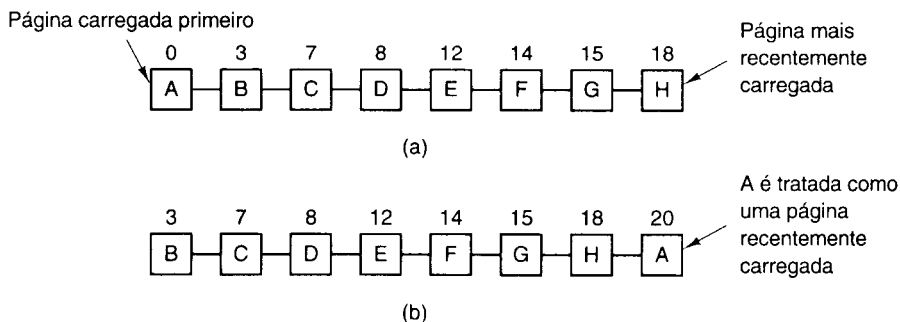



Figura 4-13 A operação de segunda chance. (a) Páginas classificadas pela ordem FIFO. (b) Lista de páginas se uma falha de página ocorrer no tempo 20 e A se tiver seu bit R ligado.

Figura 4-14 O algoritmo de substituição de página do relógio.

ritmo seja chamado de **relógio.** Ele difere do algoritmo de segunda chance somente na implementação.

4.4.6 Algoritmo de Substituição de Página menos Recentemente Utilizada (LRU)

Uma boa aproximação ao algoritmo ótimo é baseada na observação de que páginas que foram intensamente utilizadas nas últimas instruções provavelmente serão intensamente utilizadas novamente nas seguintes. Inversamente, páginas que não foram utilizadas por muito tempo provavelmente permanecerão não-utilizadas durante mais tempo. Essa idéia sugere um algoritmo realizável: quando uma falha de página ocorre, jogue fora a página que não foi utilizada por mais tempo. Essa estratégia é chamada paginação **LRU** (***Least Recently Used***, menos recentemente utilizada).

Embora o LRU seja teoricamente realizável, ele não é barato. Para implementar completamente o LRU, é necessário manter uma lista encadeada de todas as página na memória, com as páginas mais recentemente utilizadas na frente e as menos recentemente utilizadas no fundo. A dificuldade é que a lista deve ser atualizada a cada referência de memória. A operação de localizar uma página na lista, excluí-la e, então, movê-la para a frente consome muito tempo, mesmo em hardware (supondo que tal hardware pudesse ser construído).

Entretanto, há outras maneiras de implementar LRU com hardware especial. Consideraremos a maneira simples primeiro. Esse método requer que se equipe o hardware com um contador de 64 bits, C, que é automaticamente incrementado a cada instrução. Além disso, cada entrada de tabela de página também deve ter um campo suficientemente grande para conter o contador. Após cada referência de memória, o valor atual de C é armazenado na entrada da tabela de páginas referente à página que for referenciada. Quando uma falha de página ocorre, o sistema operacional examina todos os contadores na tabela de páginas para localizar o menor. Essa página é a menos recentemente utilizada.

Vejamos agora um segundo algoritmo de LRU em hardware. Para uma máquina com n molduras de página, o hardware de LRU pode manter uma matriz de $n \times n$ bits, inicialmente todos zero. Sempre que a moldura de página k é referenciada, o hardware primeiro configura todos os bits da linha k para 1, então, configura todos os bits da coluna k como 0. Em qualquer instante, a linha cujo valor binário é menor é a menos recentemente utilizada, a linha cujo valor está imediatamente acima é a próxima menos recentemente utilizada, etc. Os trabalhos desse algoritmo são dados na Figura 4-15 para quatro molduras de página e referências de página na ordem

0 1 2 3 2 1 0 3 2 3

Depois que página 0 é referenciada temos a situação da Figura 4-15(a) e assim por diante.

4.4.7 Simulação de LRU em Software

Embora os dois algoritmos de LRU anteriores sejam realizáveis a princípio, poucas máquinas, se é que alguma, têm esse hardware; então, são de pouco valor para o projetista de sistema operacional que está fazendo um sistema para uma máquina que não tem esse hardware. Em vez disso, é necessária uma solução que possa ser implementada em software. Uma possibilidade é chamada algoritmo **não freqüentemente utilizada** ou **NFU**. Ele exige um contador de software associado com cada página, inicialmente zero. Em cada interrupção de relógio, o sistema operacional varre todas as páginas na memória. Para cada página, o bit R, que é 0 ou 1, é adicionado ao contador. De fato, os contadores são uma tentativa de monitorar a freqüência com que cada página foi referenciada. Quando ocorre uma falha de página, a página com o contador mais baixo é escolhida para substituição.

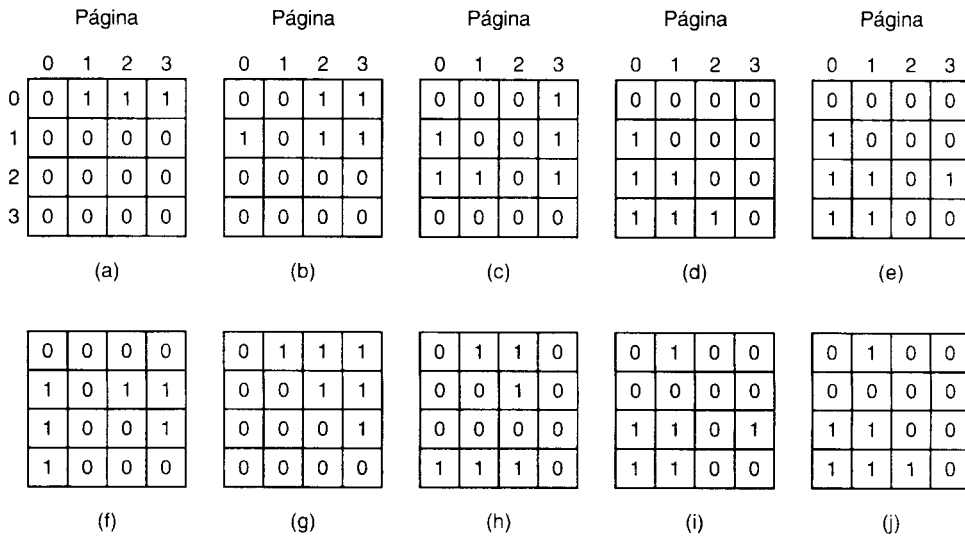



Figura 4-15 LRU utilizando uma matriz.

O problema principal com o NFU é que ele nunca se esquece de qualquer coisa. Por exemplo, em um compilador de múltiplos passos, as páginas que foram intensamente utilizadas durante o passo 1 podem ainda ter uma contagem alta bem depois dos passos seguintes. De fato, se acontecer de o passo 1 ter o tempo de execução mais longo de todos os passos, as páginas que contêm o código para os passos seguintes sempre podem ter contagens mais baixas que as páginas do passo 1. Portanto, o sistema operacional removerá páginas úteis em vez de páginas não mais em uso.

Felizmente, uma pequena modificação para o NFU permite que ele simule bem o LRU. A modificação tem duas partes. Em primeiro lugar, cada contador é deslocado 1 bit à direita antes de o bit *R* ser adicionado. Em segundo lugar, o bit *R* é adicionado ao bit mais à esquerda em vez de ao bit mais à direita.

A Figura 4-16 ilustra como o algoritmo modificado, conhecido como algoritmo de **idade**, funciona. Suponha que depois do primeiro tique de relógio os bits *R* para páginas 0 a 5 tenham os valores 1, 0, 1, 0, 1 e 1 respectivamente (página 0 é 1, página 1 é 0, página 2 é 1 e assim por diante.) Em outras palavras, entre o tique 0 e o tique 1, as páginas 0, 2, 4 e 5 foram referenciadas, configurando seus bits *R* como 1, enquanto os outros permaneceram 0. Depois que os seis contadores correspondentes foram deslocados e tiveram o bit *R* inserido à esquerda, eles têm os valores mostrados na Figura 4-16(a). As quatro colunas restantes mostram os seis contadores depois dos quatro tiques de relógio seguintes.

Quando ocorre uma falha de página, a página cujo contador é o menor é removida. É claro que uma página que não foi referenciada por, digamos, quatro tiques de relógio, terá quatro zeros começando seu contador e, assim, terá um valor menor do que um contador que não foi referenciado por três tiques de relógio.

Esse algoritmo difere do LRU sob dois aspectos. Considere as páginas 3 e 5 na Figura 4-16(e). Nenhuma foi referenciada por dois tiques de relógio; ambas foram referenciadas no tique antes disso. De acordo com o LRU, se uma página deve ser substituída, devemos escolher uma dessas duas. O problema é que não sabemos qual delas foi referenciada pela última vez no intervalo entre o tique 1 e o tique 2. Registrando apenas um bit por intervalo de tempo, perdemos a capacidade de distinguir entre referências que ocorreram mais cedo ou mais tarde no intervalo de um tique do relógio. Tudo que podemos fazer é remover a página 3, porque a página 5 também foi referenciada dois tiques antes e a página 3, não.

A segunda diferença entre o LRU e a idade é que na idade os contadores têm um número finito de bits, 8 bits nesse exemplo. Suponha que duas páginas tenham um valor de contador 0. Tudo que podemos fazer é escolher um deles aleatoriamente. Na realidade, é bem possível que uma das páginas tenha sido referenciada pela última vez 9 tiques atrás e a outra tenha sido referenciada por último 1.000 tiques atrás. Não temos nenhuma maneira de ver isso. Na prática, entretanto, 8 bits, em geral, são suficientes se um tique de relógio corresponde a algo em torno de 20ms. Se uma página não for referenciada em 160ms, provavelmente ela não é importante.

4.5 QUESTÕES DE PROJETO PARA SISTEMAS DE PAGINAÇÃO

Nas seções anteriores, explicamos como a paginação funciona e oferecemos alguns dos algoritmos de substitui-

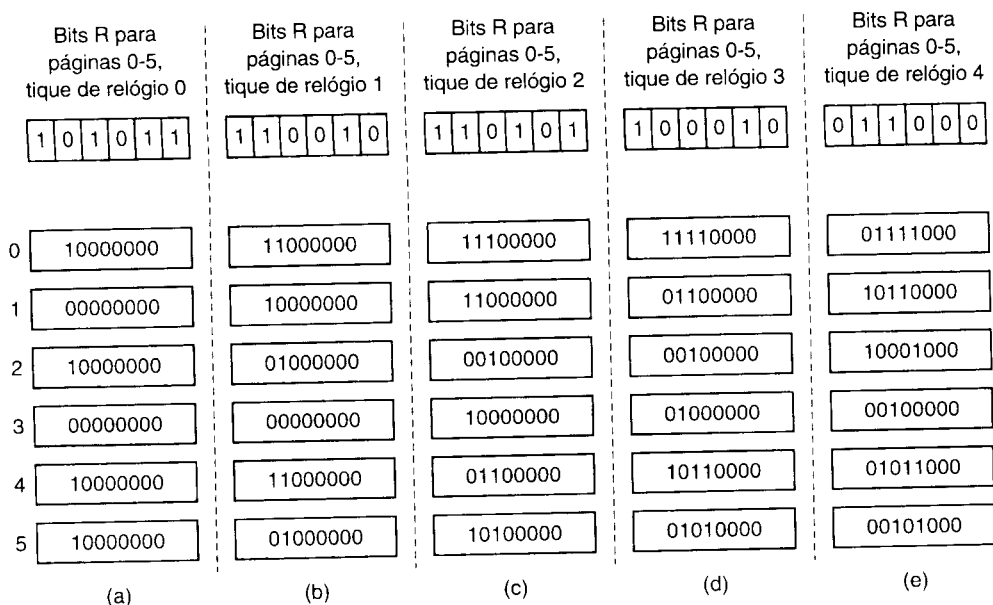



Figura 4-16 O algoritmo de idade simula LRU em software. São mostradas seis páginas para cinco tiques de relógio. Os cinco tiques de relógio são representados de (a) a (e).

ção de página básicos. Mas conhecer apenas a mecânica não é suficiente. Para projetar um sistema, você precisa conhecer muito mais para fazê-lo funcionar bem. É como a diferença entre saber mover a torre, o cavalo, o bispo e outras peças no xadrez e ser um bom jogador. Nas seções a seguir, veremos outras questões que projetistas de sistemas operacionais devem considerar cuidadosamente a fim de obter um bom desempenho de um sistema de paginação.

4.5.1 Modelo de Conjunto Funcional

Na forma mais pura de paginação, processos são iniciados sem nenhuma de suas páginas na memória. Logo que a CPU tenta buscar a primeira instrução, ela obtém uma falha de página, fazendo com que o sistema operacional traga a página que contém a primeira instrução. Outras falhas de página para variáveis globais e para a pilha, em geral, seguem-se rapidamente. Depois de um tempo, o processo tem a maioria das páginas de que necessita e estabiliza-se, executando com relativamente poucas falhas de página. Essa estratégia é chamada **paginação sob demanda** porque as páginas são carregadas somente quando são necessárias, não antes.

Naturalmente, é suficientemente fácil escrever um programa de teste que de forma sistemática leia todas as páginas em um grande espaço de endereços, causando tantas falhas de página que não haja memória suficiente para armazenar tudo. Felizmente a maioria dos processos não funciona assim. Eles apresentam uma **referência localizada**, indicando que durante qualquer fase de execução, o processo referencia somente uma fração relativamente pequena de suas páginas. Cada passagem de um compilador de múltiplas passagens, por exemplo, referencia somente uma fração de todas as páginas.

O conjunto de páginas que um processo está atualmente utilizando é chamado **conjunto funcional** (*working set*) (Denning, 1968a; Denning, 1980). Se o conjunto funcional inteiro estiver na memória, o processo executará sem causar muitas falhas até que ele entre em outra fase de execução (p. ex., a próxima passagem do compilador). Se a memória disponível for muito pequena para armazenar o conjunto funcional inteiro, o processo irá causar muitas falhas de página e executar lentamente, uma vez que a execução de uma instrução freqüentemente leva alguns nanossegundos, e a leitura em uma página do disco, em geral, leva dezenas de milissegundos. A uma velocidade de uma ou duas instruções por 20 milissegundos, ele irá demorar muito para terminar. Quando um programa causa falhas de página a cada poucas instruções, diz-se que ele está **criando lixo** (*thrashing*) (Denning, 1968b).

Em um sistema de compartilhamento de tempo, os processos são freqüentemente movidos para disco (i. e., todas as suas páginas são removidas da memória) para deixar outros processos ter vez na CPU. A pergunta que surge é saber o que fazer quando um processo é trazido de volta novamente. Tecnicamente, nada precisa ser feito. O processo simplesmente causará falhas de página até que seu conjunto funcional tenha sido carregado. O problema é que ter 20, 50 ou até 100 falhas de página cada vez que um processo é carregado é lento e também desperdiça tempo

considerável de CPU, uma vez que o sistema operacional leva alguns milissegundos de tempo da CPU para processar uma falha de página.

Portanto, muitos sistemas de paginação tentam monitorar o conjunto funcional de cada processo e certificar-se de que ele está na memória antes de deixar o processo executar. Essa abordagem é chamada **modelo do conjunto funcional** (Denning, 1970). Ele foi projetado principalmente para reduzir a taxa de falhas de página. Carregar as páginas *antes* de deixar os processos executarem também é chamado **pré-paginação.**

Para implementar o modelo do conjunto funcional, é necessário que o sistema operacional monitore quais páginas estão no conjunto funcional. Uma maneira de monitorar essas informações é utilizar o algoritmo de idade discutido acima. Qualquer página que contenha um bit 1 entre os n bits de ordem alta do contador é considerada como membro do conjunto funcional. Se uma página não for referenciada em n tiques consecutivos de relógio, ela é retirada do conjunto funcional. O parâmetro n tem de ser determinado experimentalmente para cada sistema, mas o desempenho do sistema normalmente não é especialmente sensível ao valor exato.

As informações sobre o conjunto funcional podem ser utilizadas para melhorar o desempenho do algoritmo do relógio. Normalmente, quando o ponteiro indica uma página cujo bit R é 0, a página é expulsa. O aprimoramento é verificar se essa página é parte do conjunto funcional do processo atual. Se for, a página é poupada. Esse algoritmo é chamado ***wsclock.***

4.5.2 Política de Alocação Local *Versus* Global

Nas seções precedentes, discutimos vários algoritmos para escolher uma página para substituir quando ocorre uma falha. Uma questão importante associada a essa escolha (que cuidadosamente varremos para baixo do tapete até agora) é como a memória deve ser alocada entre os processos executáveis que competem.

Dê uma olhada na Figura 4-17(a). Nessa figura, três processos, A, B e C, compõem o conjunto de processos executáveis. Suponha que A provoque uma falha de página. O algoritmo de substituição de página deve tentar localizar a página menos recentemente utilizada, considerando somente as seis páginas atualmente alocadas para A ou deve considerar todas as páginas na memória? Se olhar somente as páginas de A, a página com o valor de idade mais baixo é $A5$, então, obteremos a situação da Figura 4-17(b)

Por outro lado, se a página com o valor de idade mais baixo for removida sem considerar de quem é essa página, a página $B3$ poderá ser escolhida, e obteremos a situação da Figura 4-17(c). O algoritmo da Figura 4-17(b) é chamado algoritmo de substituição de página **local**, ao passo que o da Figura 4-17(c) é chamado algoritmo **global**. Algoritmos locais efetivamente correspondem a alocar para cada processo uma fração fixa da memória. Algoritmos globais alocam dinamicamente molduras de página entre os processos executáveis. Assim, o número de molduras de página atribuídas a cada processo varia com o tempo.

Figura 4-17 Substituição de página local *versus* global. (a) Configuração original. (b) Substituição de página de local. (c) Substituição de página global.

Em geral, algoritmos globais funcionam melhor, especialmente quando o tamanho do conjunto funcional pode variar durante o tempo de vida de um processo. Se um algoritmo local for utilizado, e o conjunto funcional crescer, resultará na criação de lixo, mesmo que haja abundância de molduras de página livres. Se o conjunto funcional encolher, algoritmos locais desperdiçam memória. Se um algoritmo global for utilizado, o sistema deve continuamente decidir quantas molduras de página atribuir a cada processo. Uma maneira é monitorar o tamanho do conjunto funcional como indicado pelos bits de idade, mas essa abordagem necessariamente não previne a criação de lixo. O conjunto funcional pode alterar o tamanho em microssegundos, ao passo que os bits de idade são uma medida bruta distribuída sobre um número de tiques de relógio.

Outra abordagem é ter um algoritmo para alocar molduras de página para processos. Uma maneira é periodicamente determinar o número de processos em execução e alocar para cada processo uma parte igual. Assim com 475 molduras de página disponíveis (i. e., não-pertencentes ao sistema operacional) e 10 processos, cada processo obtém 47 molduras. As 5 restantes são reservadas para serem utilizadas quando ocorrerem falhas de página.

Embora esse método pareça justo, faz pouco sentido dar partes iguais da memória para um processo de 10K e para um processo de 300K. Em vez disso, as páginas podem ser alocadas na proporção do tamanho total de cada processo, com um processo de 300K obtendo 30 vezes a cota de um processo de 10K. Provavelmente é inteligente dar algum número mínimo a cada processo, para que ele possa executar independentemente de quão pequeno ele seja. Em algumas máquinas, por exemplo, uma única instrução pode necessitar de até seis páginas porque a própria instrução, o operando da origem e o operando de destino, podem ultrapassar os limites da página. Com a alocação de somente cinco páginas, programas que contenham essas instruções não poderão executar.

Nem a alocação igualitária nem o método de alocação proporcional lidam diretamente com o problema de criação de lixo. Uma maneira mais direta de controlar isso é utilizar o algoritmo de alocação por **Freqüência de Falha de Página (*Page Fault Frequency – PFF*)**. Para uma classe extensa de algoritmos de substituição de página, incluindo LRU, sabe-se que a taxa de falhas diminui à medida que mais páginas são alocadas, como já discutimos. Essa propriedade é ilustrada na Figura 4-18.

A linha tracejada *A* corresponde a uma taxa de falhas de página que é inaceitavelmente alta, então, o processo que falha recebe mais molduras de página para reduzir a taxa de falhas. A linha tracejada *B* corresponde a uma taxa de falhas de página tão baixa que se pode concluir que o processo tem memória demais. Nesse caso, algumas molduras de página podem ser tiradas dele. Assim, o PFF tenta manter a taxa de paginação dentro de limites aceitáveis.

Se o PFF descobre que há tantos processos na memória que não é possível mantê-los todos abaixo de *A*, algum processo é removido da memória e suas molduras de página são divididas entre os processos que permanecem ou são colocados em uma reserva de páginas disponíveis que pode ser utilizada em falhas de página subseqüentes. A decisão de remover um processo da memória é uma forma de controle de carga. Isso mostra que mesmo com paginação, a comutação em disco ainda é necessária, só que agora é utilizada para reduzir a exigência potencial para a memória, em vez de reivindicar blocos para utilização imediata.

4.5.3 Tamanho de Página

O tamanho de página é freqüentemente um parâmetro que pode ser escolhido pelo sistema operacional. Mesmo se o hardware for projetado com, por exemplo, páginas de 512 bytes, o sistema operacional facilmente pode considerar as páginas 0 e 1, 2 e 3, 4 e 5 e assim por diante, como páginas de 1K, alocando sempre duas molduras de página de 512 bytes consecutivos para elas.

Determinar o tamanho ótimo de página exige balancear vários fatores que competem entre si. Para começar, um segmento de texto de dados ou de pilha escolhido aleatoriamente não ocupará um número integral de páginas.

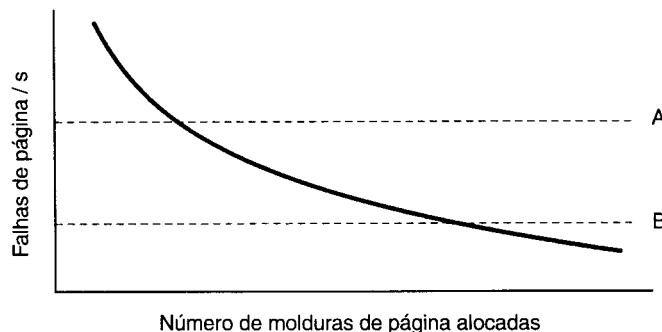



Figura 4-18 Taxa de falhas de página como uma função do número de molduras de página alocadas.

Na média, metade da página final estará vazia. O espaço extra naquela página é desperdiçado. Esse desperdício é denominado **fragmentação interna.** Com n segmentos na memória e um tamanho de página de p bytes, $np/2$ bytes serão desperdiçados na fragmentação interna. Esse raciocínio argumenta em favor de um tamanho de página pequeno.

Outro argumento para um tamanho de página pequeno torna-se evidente se pensamos sobre um programa consistindo em oito fases seqüenciais de 4K cada. Com um tamanho de página 32K, deve-se alocar ao programa 32K todo o tempo. Com um tamanho de página de 16K, é necessário somente 16K. Com um tamanho de página de 4K ou menor, exige-se apenas 4K em qualquer instante. Em geral, um tamanho de página grande fará com que os programas menos utilizados permaneçam mais na memória que com um tamanho de página pequeno.

Por outro lado, páginas pequenas significam que os programas necessitarão de muitas páginas, daí uma tabela de páginas grande. Um programa de 32K precisa somente de quatro páginas de 8K, mas de 64 páginas de 512 bytes. As transferências para e do disco são geralmente uma página por vez, sendo a maior parte do tempo dedicada para a busca e para o retardo rotacional, de modo que transferir uma página pequena leva quase tanto tempo quanto transferir uma página grande. Talvez leve 64×15ms para carregar 64 páginas de 512 bytes, mas somente 4×25ms para carregar quatro páginas de 8K.

Em algumas máquinas, a tabela de páginas deve ser carregada em registradores de hardware cada vez que a CPU alterna de um processo para outro. Nessas máquinas, ter um tamanho de página pequeno significa que o tempo solicitado para carregar os registradores de página fica mais longo conforme o tamanho de página fica menor. Além disso, o espaço ocupado pela tabela de páginas aumenta conforme diminui o tamanho de página.

Este último ponto pode ser analisado matematicamente. Suponha que o tamanho médio de um processo seja s bytes e que o tamanho de página seja p bytes. Além disso, suponha que cada entrada de página exija e bytes. O número aproximado de páginas necessário por processo é então s/p, ocupando se/p bytes de espaço da tabela de páginas. A memória desperdiçada na última página do processo devido à fragmentação interna é $p/2$. Portanto, o *overhead* total devido à tabela de páginas e à perda por fragmentação interna é dado por

$$\text{overhead} = se/p + p/2$$

O primeiro termo (tamanho da tabela de páginas) é grande quando o tamanho de página é pequeno. O segundo termo (fragmentação interna) é grande quando o tamanho de página é grande. O ótimo deve estar em algum lugar entre esses. Fazendo a derivada primeira em relação à p e igualando a zero, obtemos a equação

$$-se/p^2 + 1/2 = 0$$

Dessa equação, podemos derivar uma fórmula que dá o tamanho ótimo de página (considerando somente a memória desperdiçada na fragmentação e o tamanho da tabela de páginas). O resultado é:

$$p = \sqrt{2se}$$

Para s = 128K e e = 8 bytes por entrada da tabela de páginas, o tamanho ótimo de página é 1448 bytes. Na prática, 1K ou 2K seriam utilizados, dependendo dos outros fatores (p. ex., velocidade do disco). Muitos computadores disponíveis comercialmente utilizam tamanhos de página que variam de 512 bytes a 64K.

4.5.4 Interface de Memória Virtual

Até agora, nossa conversa inteira assumiu que a memória virtual é transparente para processos e programadores, quer dizer, tudo que eles vêem é um grande espaço de endereço virtual em um computador com uma memória física pequena (menor). Com muitos sistemas, isso é verdadeiro, mas em alguns sistemas avançados, os programadores têm algum controle sobre o mapa de memória e podem empregá-lo de modos não-tradicionais. Nesta seção, veremos resumidamente alguns deles.

Uma razão para dar aos programadores controle sobre seu mapa de memória é permitir que dois ou mais processos compartilhem a mesma memória. Se os programadores puderem atribuir nomes a regiões de sua memória, pode ser possível para um processo dar o nome de uma região de memória para outro processo de modo que este também possa mapeá-la. Com dois (ou mais) processos compartilhando as mesmas páginas, torna-se possível o compartilhamento de grande largura de banda — um processo grava na memória compartilhada e outro lê a partir dela.

O compartilhamento de páginas também pode ser utilizado para implementar um sistema de passagem de mensagens de alto desempenho. Normalmente, quando mensagens são passadas, os dados são copiados de um espaço de endereço para outro, a um custo considerável. Se os processos puderem controlar seu mapa de páginas, uma mensagem pode ser passada, fazendo o processo remetente desmapear a(s) página(s) que contém(êm) a mensagem, e o processo receptor mapeá-las. Aqui somente os nomes de página têm de ser copiados, em vez de todos os dados.

Ainda outra técnica avançada de gerenciamento de memória é a **memória compartilhada distribuída** (Feeley *et al.*, 1995; Li e Hudak, 1989; Zekauskas *et al.*, 1994). A idéia aqui é permitir que múltiplos processos sobre uma rede compartilhem um conjunto de páginas, possivelmente, mas não necessariamente, como um único espaço de endereço linear compartilhado. Quando um processo referencia uma página que atualmente não está mapeada, ele obtém uma falha de página. O manipulador de falha de página, que pode estar no *kernel* ou no espaço de usuário, então, localiza a máquina que armazena a página e envia uma mensagem solicitando que ele desmapeie a página e

envie pela rede. Quando a página chega, ela está mapeada, e a instrução que está falhando é reiniciada.

4.6 SEGMENTAÇÃO

A memória virtual discutida até agora é unidimensional porque os endereços virtuais vão de 0 a alguns endereços máximos, um endereço depois do outro. Para muitos problemas, ter dois ou mais espaços de endereço virtuais separados é muito melhor do que ter apenas um. Por exemplo, um compilador tem muitas tabelas que são construídas durante a compilação, possivelmente incluindo

1. O texto fonte sendo salvo para a listagem impressa (em sistemas de lote).
2. A tabela de símbolos, contendo os nomes e os atributos de variáveis.
3. A tabela contendo todas as constante de número inteiro e de ponto flutuante utilizadas.
4. A árvore de análise, contendo a análise sintática do programa.
5. A pilha utilizada para chamadas de procedimento dentro do compilador.

Cada uma das primeiras quatro tabelas cresce continuamente à medida que a compilação procede. A última cresce e encolhe de modo imprevisível durante a compilação. Em uma memória unidimensional, essas cinco tabelas teriam de receber porções contíguas do espaço de endereço virtual, como na Figura 4-19.

Considere o que acontece se um programa tem um número excepcionalmente grande de variáveis. O espaço de endereço alocado para a tabela de símbolos pode ser totalmente preenchido, mas pode haver muito espaço nas outras tabelas. O compilador poderia, naturalmente, simplesmente emitir uma mensagem dizendo que a compilação não continuou devido a variáveis demais, mas fazer isso não parece muito esportivo quando espaço não-utilizado é deixado em outras tabelas.

Outra possibilidade é brincar de Robin Hood, roubando espaço das tabelas com um excesso de espaço e dando às tabelas com espaço pequeno. Essa troca pode ser feita, mas é análoga a gerenciar seus próprios *overlays* — um aborrecimento no melhor dos casos, e um trabalho muito tedioso e desvantajoso no pior.

O que é realmente necessário é uma maneira de liberar o programador de gerenciar a expansão e a contração de tabelas, da mesma maneira que a memória virtual elimina a preocupação de organizar o programa em *overlays*.

Uma solução extremamente simples e direta é prover a máquina com muitos espaços de endereço completamente independentes, denominados **segmentos.** Cada segmento consiste em uma seqüência linear de endereços, de 0 até algum máximo. O comprimento de cada segmento pode ser qualquer coisa de 0 até o máximo permitido. Segmentos diferentes podem e normalmente têm comprimentos diferentes. Além disso, os comprimentos dos segmentos podem variar durante a execução. O comprimento de um segmento de pilha pode ser aumentado sempre que algo é colocado na pilha e diminuído sempre que algo é retirado da pilha.

Porque cada segmento constitui um espaço de endereço distinto, segmentos diferentes podem crescer ou podem encolher independentemente, sem afetar um ao outro. Se uma pilha em um certo segmento precisar de mais espaço de endereço para crescer, pode tê-lo, porque não há nada

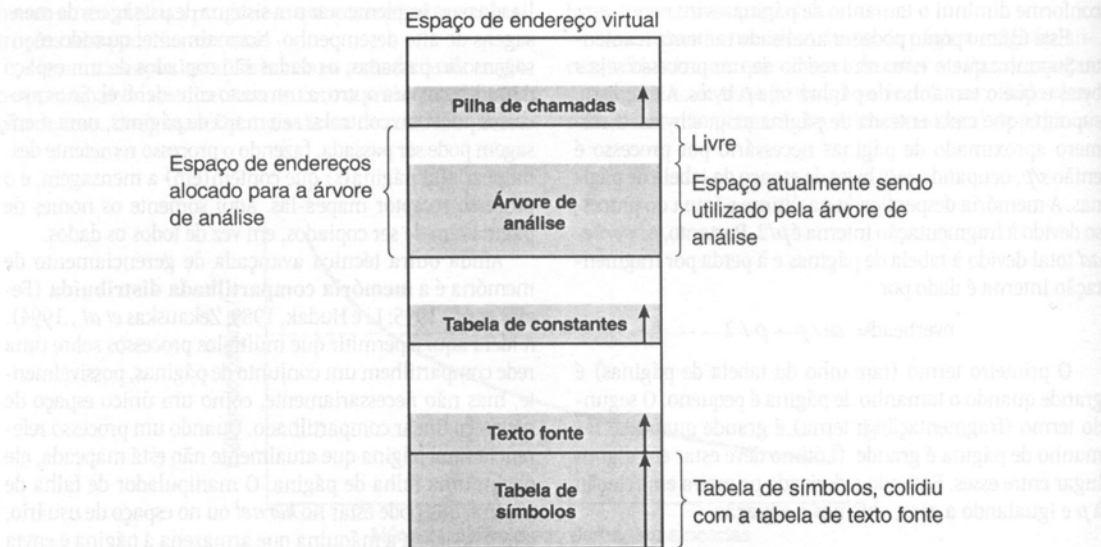



Figura 4-19 Em um espaço unidimensional de endereços com tabelas crescentes, uma tabela pode colidir com outra.

mais em seu espaço de endereço em que colidir. Naturalmente, um segmento pode encher-se, mas os segmentos são normalmente muito grandes, portanto, essa ocorrência é rara. Para especificar um endereço nessa memória segmentada ou bidimensional, o programa deve fornecer um endereço de duas partes, um número de segmento e um endereço dentro do segmento. A Figura 4-20 ilustra uma memória segmentada sendo utilizada pelas tabelas de compilador discutidas anteriormente.

Realçamos que um segmento é uma entidade lógica, da qual o programador está ciente e à qual ele utiliza como uma entidade lógica. Um segmento pode conter um procedimento ou uma matriz, ou uma pilha ou uma coleção de variáveis escalares, mas normalmente ele não contém uma mistura de tipos diferentes.

Uma memória segmentada tem outras vantagens além de simplificar o tratamento de estruturas de dados que estão crescendo ou encolhendo. Se cada procedimento ocupar um segmento separado, com o endereço 0 como seu endereço inicial, a linkedição de procedimentos compilados separadamente é muitíssimo simplificada. Depois que todos os procedimentos que constituem um programa foram compilados e linkeditados, uma chamada de procedimento para o procedimento no segmento n utilizará o endereço de duas partes (n, 0) para endereçar a palavra 0 (o ponto de entrada).

Se o procedimento no segmento n for subseqüentemente modificado e recompilado, nenhum outro procedimento precisará ser alterado (porque nenhum dos endereços iniciais foi modificado), mesmo se a nova versão for maior do que a antiga. Com uma memória unidimensional, os procedimentos são empacotados um ao lado do outro, sem espaço de endereço entre eles. Portanto, alterar o tamanho de um procedimento pode afetar o endereço inicial dos outros procedimentos não-relacionados. Isso, por sua vez, exige modificar todos os procedimentos que chamam qualquer um dos procedimentos movidos, para incorporar seus novos endereços iniciais. Se um programa contiver centenas de procedimentos, tal processo pode ser caro.

A segmentação também facilita compartilhar procedimentos ou dados entre vários processos. Um exemplo comum é a **biblioteca compartilhada.** Estações de trabalho modernas que executam avançados sistemas de janelas, com freqüência, têm bibliotecas gráficas extremamente grandes compiladas em quase cada programa. Em um sistema segmentado, a biblioteca gráfica pode ser colocada em um segmento e compartilhada por múltiplos processos, eliminando a necessidade de tê-la no espaço de endereço de cada processo. Embora também seja possível ter bibliotecas compartilhadas em sistemas de paginação puros, isso é muito mais complicado. De fato, esses sistemas fazem isso simulando segmentação.

Como cada segmento forma uma entidade lógica de que o programador está ciente, tal como um procedimento, uma matriz ou uma pilha, segmentos diferentes podem ter tipos de proteção diferentes. Um segmento de procedimento pode ser especificado como de execução somente, sendo proibido ler dele ou armazenar nele. Uma matriz de ponto flutuante pode ser especificada como de leitura/gravação, mas não de execução, e as tentativas de saltar para ela serão interceptadas. Esse tipo de proteção é útil para detectar erros de programação.

Você deve tentar entender por que a proteção faz sentido em uma memória segmentada, mas não em uma memória paginada unidimensional. Em uma memória segmentada, o usuário está ciente do que está em cada segmento. Normalmente, um segmento não conteria um procedimento e uma pilha, por exemplo, mas um ou outro.

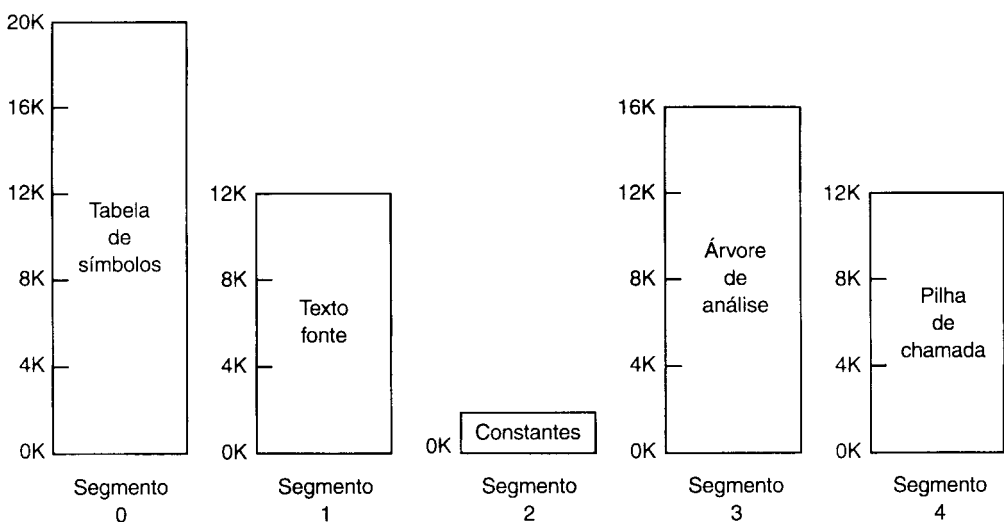



Figura 4-20 Uma memória segmentada permite que cada tabela cresça ou encolha independentemente das outras tabelas.

Uma vez que cada segmento contém somente um tipo de objeto, o segmento pode ter a proteção apropriada para esse tipo particular. A paginação e a segmentação são comparadas na Figura 4-21.

O conteúdo de uma página é, em certo sentido, acidental. O programador ignora até mesmo o fato de que a paginação ocorre. Embora colocar alguns bits em cada entrada da tabela de página para especificar o acesso permitido seja possível, para utilizar esse recurso o programador teria de monitorar onde os limites de página estão em seu espaço de endereço. Isso é precisamente o tipo de administração para o qual a paginação foi inventada para eliminar. Como o usuário de uma memória segmentada tem a ilusão de que todos os segmentos estão na memória principal o tempo todo — isto é, é possível endereçá-los como se estivessem — ele pode proteger cada segmento separadamente, sem se preocupar com a administração de *overlays*.

4.6.1 Implementação da Segmentação Pura

A implementação de segmentação difere da paginação em uma maneira essencial: as páginas têm tamanhos fixos, os segmentos, não. A Figura 4-22(a) mostra um exemplo de memória física inicialmente contendo cinco segmentos. Agora considere o que acontece se o segmento 1 for expulso e o segmento 7, que é menor, for colocado em seu lugar. Chegamos à configuração de memória da Figura 4-22(b). Entre o segmento 7 e o segmento 2, há uma área não-utilizada — isto é, uma lacuna. Então, o segmento 4 é substituído pelo segmento 5, como na Figura 4-22(c), e o segmento 3 é substituído pelo segmento 6, como na Figura 4-22(d) depois que o sistema tiver executado temporariamente, a memória será dividida em um número de trechos, alguns contendo segmentos e outros contendo lacunas. Esse fenômeno, chamado ***checkerboarding*** (formação de um tabuleiro de xadrez) ou **fragmentação externa**, desperdiça memória nas lacunas. Isso pode ser tratado com compactação, como mostrado na Figura 4-22(e).

4.6.2 Segmentação com Paginação: MULTICS

Se os segmentos são grandes, pode ser inconveniente, ou mesmo impossível, mantê-los na memória principal em sua totalidade. Isso leva à idéia de paginá-los, de modo que somente as página que realmente são exigidas precisem ficar por perto. Vários sistemas importantes suportaram segmentos paginados. Nesta seção, descreveremos o primeiro: MULTICS. Na próxima, discutiremos um mais recente: o Pentium da Intel.

O MULTICS rodava nas máquinas Honeywell 6000 e em seus descendentes e fornecia a cada programa uma memória virtual de até 2^{18} segmentos (mais de 250.000), cada um deles podendo ter o comprimento de até 65.536 palavras (de 36 bits). Para implementar isso, os projetistas do MULTICS escolheram tratar cada segmento como uma memória virtual e paginá-lo, combinando as vantagens da paginação (tamanho de página uniforme e não ter de manter o segmento inteiro na memória se ao menos parte

Consideração	Paginação	Segmentação
O programador precisa estar ciente de que essa técnica está sendo utilizada?	Não	Sim
Quantos espaços lineares de endereço existem?	1	Muitos
O total de espaço de endereço pode exceder ao tamanho da memória física?	Sim	Sim
Os procedimentos e os dados podem ser diferenciados e separadamente protegidos?	Não	Sim
As tabelas cujo tamanho varia podem ser acomodadas facilmente?	Não	Sim
O compartilhamento de procedimentos entre usuários é facilitado?	Não	Sim
Por que essa técnica foi inventada?	Para obter um grande espaço linear de endereço sem precisar adquirir mais memória física	Para permitir que programas e dados sejam divididos em espaços de endereços logicamente independentes e ajudar no compartilhamento e na proteção

Figura 4-21 A comparação de paginação e de segmentação.

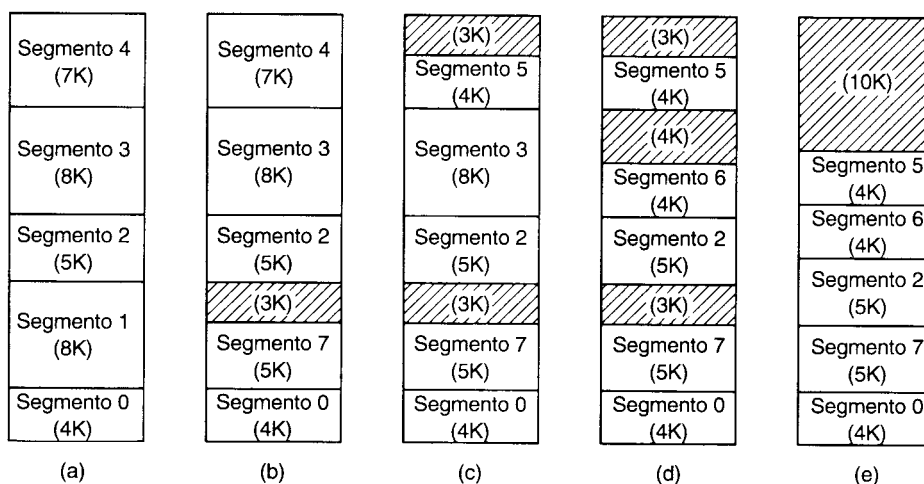



Figura 4-22 (a)-(d) Desenvolvimento da fragmentação externa. (e) Remoção da fragmentação externa por compactação.

dele estivesse sendo utilizada) com as vantagens da segmentação (facilidade de programação, modularidade, proteção e compartilhamento).

Cada programa MULTICS tem uma tabela de segmentos, com um descritor por segmento. Uma vez que há potencialmente mais que um quarto de milhão de entradas na tabela, a tabela de segmentos é, ela própria, um segmento e é paginada. Um descritor de segmento contém uma indicação se o segmento está na memória principal ou não. Se qualquer parte do segmento estiver na memória, o segmento é considerado como estando na memória e sua tabela de páginas estará na memória. Se o segmento estiver na memória, seu descritor conterá um ponteiro de 18 bits para sua tabela de páginas [veja a Figura 4-23(a) na página 238]. Como os endereços físicos são de 24 bits e as páginas são alinhadas em limites de 64 bytes (o que implica que os 6 bits de ordem inferior dos endereços de página são 000000), somente 18 bits são exigidos no descritor para armazenar um endereço de tabela de página. O descritor também contém o tamanho do segmento, os bits de proteção e alguns outros itens. A Figura 4-23(b) (ver página 238) ilustra um descritor de segmento do MULTICS. O endereço do segmento na memória secundária não está no descritor de segmento, mas em outra tabela utilizada pelo manipulador de falhas de segmento.

Cada segmento é um espaço de endereço virtual comum e é paginado da mesma maneira como a memória paginada não-segmentada descrita anteriormente neste capítulo. O tamanho normal de página é 1024 palavras (apesar de alguns segmentos pequenos utilizados pelo próprio MULTICS não serem paginados ou serem paginados em unidades de 64 palavras para economizar memória física).

Um endereço no MULTICS consiste de duas partes: o segmento e o endereço dentro do segmento. O endereço dentro do segmento é dividido ainda em um número de página e uma palavra dentro da página, como mostrado na Figura 4-24 (ver página 239). Quando uma referência de memória ocorre, o seguinte algoritmo é executado.

1. O número do segmento é utilizado para localizar o descritor do segmento.
2. Uma verificação é feita para ver se a tabela de páginas do segmento está na memória. Se a tabela de páginas estiver na memória, ela será localizada. Se não estiver, uma falha de segmento ocorre. Se houver uma violação de proteção, uma falha ocorre.
3. A entrada na tabela de páginas para a página virtual solicitada é examinada. Se a página não estiver na memória, uma falha de página ocorre. Se estiver na memória, o endereço da memória principal referente ao início da página é obtido a partir da entrada na tabela das páginas.
4. O deslocamento é adicionado à origem da página para fornecer o endereço da memória principal onde a palavra está localizada.
5. A leitura ou o armazenamento por fim acontece.

Esse processo é ilustrado na Figura 4-25 (ver página 239). Para simplificar, o fato de que o próprio segmento descritor é paginado foi omitido. O que realmente acontece é que um registrador (o registrador de base do descritor) é utilizado para localizar a tabela de páginas do segmento descritor, que, por sua vez, aponta para as páginas do segmento descritor. Uma vez que o descritor para o segmento necessário foi localizado, o endereçamento procede como mostrado na Figura 4-25.

Como você não tem nenhuma dúvida até agora, se o algoritmo precedente realmente fosse executado pelo sistema operacional em cada instrução, os programas não executariam muito rapidamente. Na realidade, o hardware do

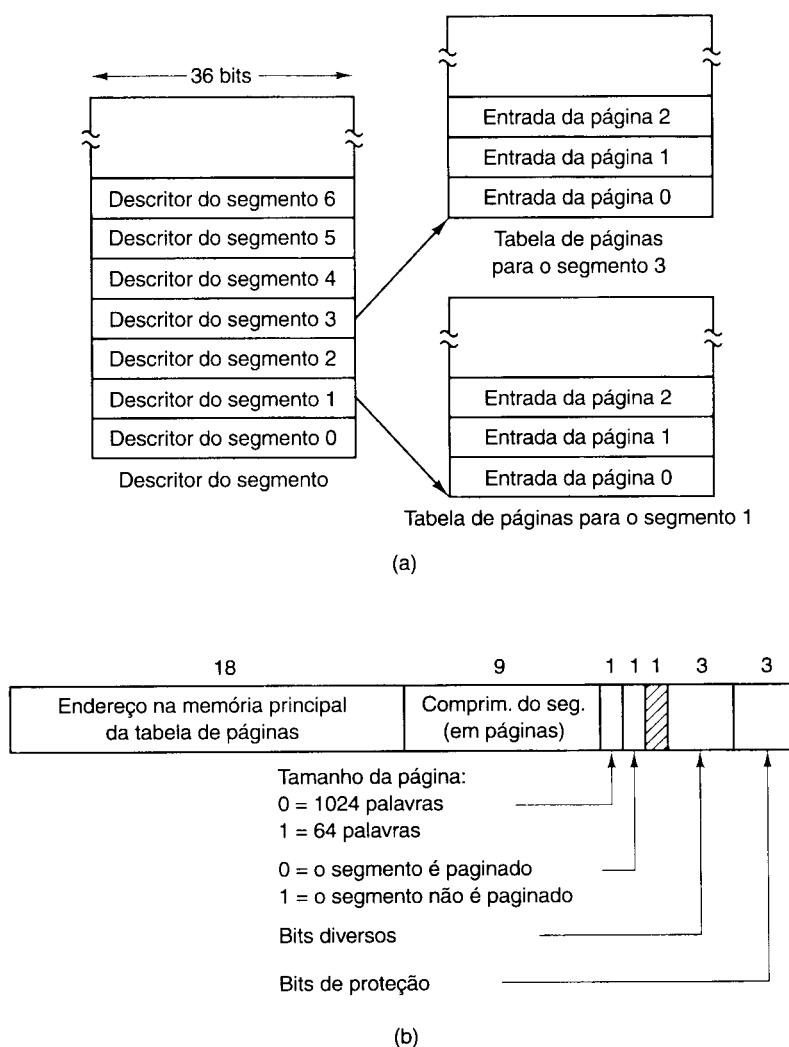



Figura 4-23 A memória virtual do MULTICS. (a) O segmento descritor aponta para as tabelas de páginas. (b) Um descritor de segmento. Os números são os comprimentos dos campos.

MULTICS contém um TLB de alta velocidade de 16 palavras que pode pesquisar todas as suas entradas em paralelo por uma dada chave. Isso é ilustrado na Figura 4-26. Quando um endereço é apresentado para o computador, o hardware de endereçamento primeiro verifica se o endereço virtual está no TLB. Se estiver, ele obtém o número na moldura de página diretamente do TLB e forma o endereço real da palavra referenciada sem olhar no segmento descritor ou na tabela de páginas.

Os endereços das 16 páginas referenciadas mais recentemente são mantidos no TLB. Os programas cujo conjunto funcional é menor do que o TLB entrarão em equilíbrio com os endereços do conjunto funcional inteiro no TLB e, portanto, executarão eficientemente. Se a página não estiver no TLB, o descritor e as tabelas de páginas realmente são referenciados para localizar o endereço da moldura de páginas e o TLB é atualizado para incluir essa página, sendo a página menos recentemente utilizada jogada fora. O campo de idade monitora qual entrada é a menos recentemente utilizada. A razão por que um TLB é utilizado é para comparar o segmento e o número de página de todas as entradas em paralelo.

4.6.3 Segmentação com Paginação: o Pentium Intel

De várias maneiras, a memória virtual do Pentium (e do Pentium Pro) assemelha-se à do MULTICS, incluindo a

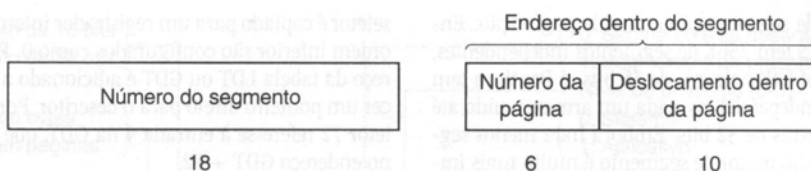



Figura 4-24 Um endereço virtual de 34 bits do MULTICS.

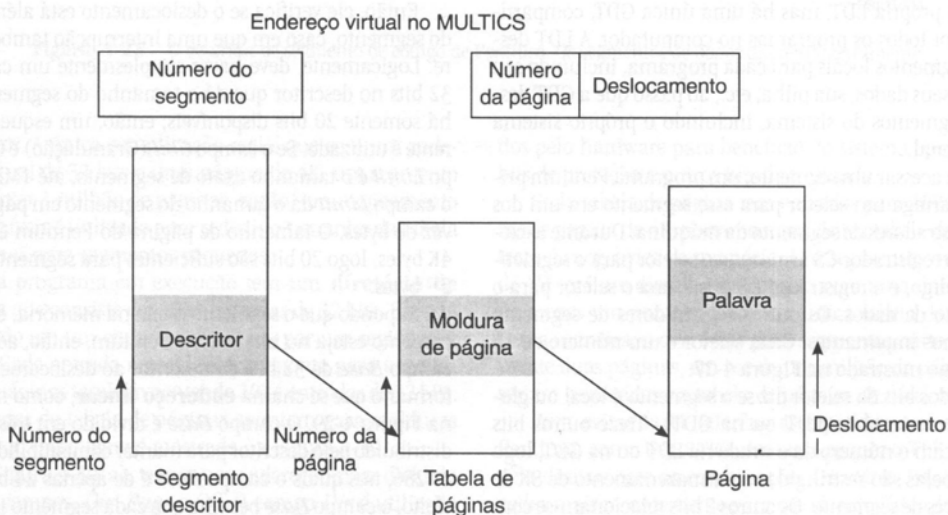



Figura 4-25 A conversão de um endereço de duas partes do MULTICS em um endereço na memória principal.

Campo de comparação

Número do segmento	Página virtual	Moldura de página	Proteção	Idade	Essa entrada está em uso?
4	1	7	Leitura/gravação	13	1
6	0	2	Somente leitura	10	1
12	3	1	Leitura/gravação	2	1
					0
2	1	0	Somente execução	7	1
2	2	12	Somente execução	9	1

Figura 4-26 Uma versão simplificada do TLB do MULTICS. A existência de dois tamanhos de página torna o TLB real mais complicado.

presença tanto de segmentação como de paginação. Enquanto o MULTICS tem 256K de segmentos independentes, cada um com até 64K palavras de 36 bits, o Pentium tem 16K segmentos independentes, cada um armazenando até 1 bilhão de palavras de 32 bits. Embora haja menos segmentos, o tamanho maior de segmento é muito mais importante, uma vez que poucos programas precisam de mais de 1.000 segmentos, mas muitos programas precisam de segmentos que armazenem megabytes.

O coração da memória virtual do Pentium consiste em duas tabelas, a **LDT (*Local Descriptor Table*, Tabela Local de Descritores)** e a **GDT (*Global Descriptor Table*, Tabela Global de Descritores)**. Cada programa tem sua própria LDT, mas há uma única GDT, compartilhada por todos os programas no computador. A LDT descreve segmentos locais para cada programa, incluindo seu código, seus dados, sua pilha, etc., ao passo que a GDT descreve segmentos do sistema, incluindo o próprio sistema operacional.

Para acessar um segmento, um programa Pentium primeiro carrega um seletor para esse segmento em um dos seis registradores de segmento da máquina. Durante a execução, o registrador CS armazena o seletor para o segmento de código, e o registrador DS armazena o seletor para o segmento de dados. Os outros registradores de segmento são menos importantes. Cada seletor é um número de 16 bits, como mostrado na Figura 4-27.

Um dos bits do seletor diz se o segmento é local ou global (i. e., se está na LDT ou na GDT). Treze outros bits especificam o número da entrada na LDT ou na GDT, logo essas tabelas são restringidas ao armazenamento de 8K de descritores de segmento. Os outros 2 bits relacionam-se com proteção e serão descritos mais tarde. O descritor 0 é proibido. Ele pode ser seguramente carregado em um registrador de segmento para indicar que o registrador de segmento não está atualmente disponível. Se utilizado, causa uma interrupção.

No momento em que um seletor é carregado em um registrador de segmento, o descritor correspondente é buscado na LDT ou na GDT e armazenado em registradores de microprograma, para que possa ser acessado rapidamente. Um descritor consiste em 8 bytes, incluindo o endereço de base, o tamanho e outras informações do segmento, como representado na Figura 4-28.

O formato do seletor foi inteligentemente escolhido para facilitar a localização do descritor. Primeiro a LDT ou a GDT é selecionada, com base no bit 2 do seletor. Então, o seletor é copiado para um registrador interno, e os 3 bits de ordem inferior são configurados como 0. Por fim, o endereço da tabela LDT ou GDT é adicionado a ele para fornecer um ponteiro direto para o descritor. Por exemplo, o seletor 72 refere-se à entrada 9 na GDT, que está localizada no endereço GDT + 72.

Vamos rastrear os passos por meio dos quais um par (seletor, deslocamento) é convertido em um endereço físico. Logo que o microprograma sabe qual registrador de segmento está sendo utilizado, ele pode localizar o descritor completo correspondente a esse seletor em seus registradores internos. Se o segmento não existe (seletor 0), ou está atualmente paginado fora, ocorre uma interrupção.

Então, ele verifica se o deslocamento está além do fim do segmento, caso em que uma interrupção também ocorre. Logicamente, deve haver simplesmente um campo de 32 bits no descritor que dá o tamanho do segmento, mas há somente 20 bits disponíveis; então, um esquema diferente é utilizado. Se o campo *Gbit* (Granulação) é 0, o campo *Limit* é o tamanho exato de segmento, até 1MB. Se é 1, o campo *Limit* dá o tamanho do segmento em páginas em vez de bytes. O tamanho de página do Pentium é fixo em 4K bytes, logo 20 bits são suficientes para segmentos de até 2^{32} bytes.

Supondo que o segmento esteja na memória, e o deslocamento esteja no intervalo, o Pentium, então, adiciona o campo *Base* de 32 bits do descritor ao deslocamento para formar o que se chama **endereço linear**, como mostrado na Figura 4-29. O campo *Base* é dividido em três partes e distribuído pelo descritor para manter compatibilidade com os 286, nos quais o campo *Base* é de apenas 24 bits. Com efeito, o campo *Base* permite que cada segmento inicie em um lugar arbitrário dentro do espaço de endereço linear de 32 bits.

Se a paginação for desativada (por um bit em um registrador de controle global), o endereço linear é interpretado como o endereço físico e enviado para a memória para leitura ou para gravação. Assim, com a paginação desativada, temos um esquema de segmentação puro, com o endereço de base de cada segmento dado em seu descritor. Os segmentos podem sobrepor-se, casualmente, provavelmente porque seria problema demais e tomaria muito tempo verificar se todos eles estariam disjuntos.

Por outro lado, se a paginação estiver ativada, o endereço linear será interpretado, como um endereço virtual e mapeado para o endereço físico, utilizando as tabelas de páginas, de maneira muito parecida com nossos exemplos

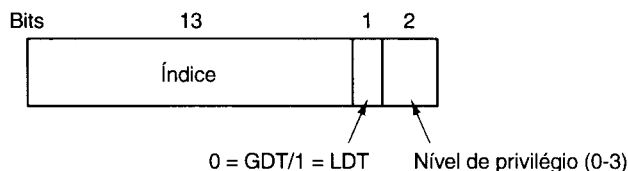



Figura 4-27 Um seletor do Pentium.

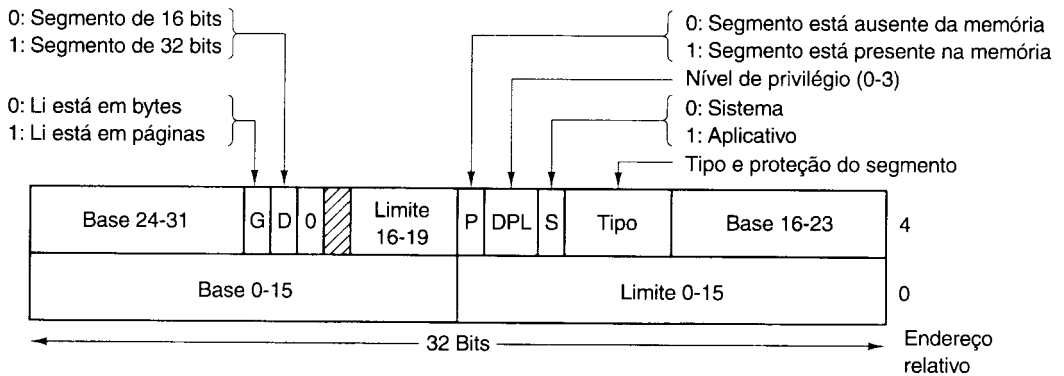



Figura 4-28 O descritor de segmento de código de Pentium. Os segmentos de dados diferem ligeiramente.

anteriores. A única complicação real é que com um endereço virtual de 32 bits e uma página de 4K, um segmento pode conter 1 milhão de páginas, então, um mapeamento de dois níveis é utilizado para reduzir o tamanho da tabela de páginas para segmentos pequenos.

Cada programa em execução tem um **diretório de páginas** que consiste em 1024 entradas de 32 bits. Ele está localizado em um endereço apontado por um registrador global. Cada entrada nesse diretório aponta para uma tabela de páginas também contendo 1024 entradas de 32 bits. As entradas da tabela de páginas apontam para molduras de página. O esquema é mostrado na Figura 4-30.

Na Figura 4-30(a), vemos um endereço linear dividido em três campos, *Dir*, *Page* e *Off*. O campo *Dir* é utilizado como índice no diretório de páginas a fim de localizar um ponteiro para a tabela de páginas adequada. Então, o campo *Page* é utilizado como um índice na tabela de páginas para localizar o endereço físico da moldura de páginas. Por fim, *Off* é adicionado ao endereço da moldura de página para obter o endereço físico do byte ou da palavra necessária.

As entradas da tabela de páginas são de 32 bits cada, 20 dos quais contêm um número de moldura de página. Os bits restantes contêm bits de acesso e bits sujos, configurados pelo hardware para benefício do sistema operacional, bits de proteção e outros bits de utilidade.

Cada tabela de página tem entradas para 1024 molduras de página de 4K; portanto, uma única tabela de páginas trata 4 megabytes de memória. Um segmento com menos de 4M terá um diretório de páginas com uma única entrada e com um ponteiro para sua única tabela de páginas. Dessa maneira, o *overhead* para segmentos curtos é de somente duas páginas, em vez de um milhão de páginas que seriam requeridas na tabela de páginas de nível único.

Para evitar fazer referências repetidas à memória, o Pentium, como o MULTICS, tem um pequeno TLB que mapeia diretamente as combinações *Diretório-Página* utilizadas mais recentemente para o endereço físico da moldura de página. Somente quando a combinação atual não está presente no TLB é que o mecanismo da Figura 4-30 é realmente executado e o TLB atualizado.

Pensando um pouco, descobrimos que o fato de que quando a paginação é utilizada, não há realmente nenhum motivo para ter o campo *Base* no descritor como diferente de zero. Tudo que o campo *Base* faz é causar um pequeno deslocamento para utilizar uma entrada no meio do diretório de páginas, em vez de no começo. A verdadeira razão

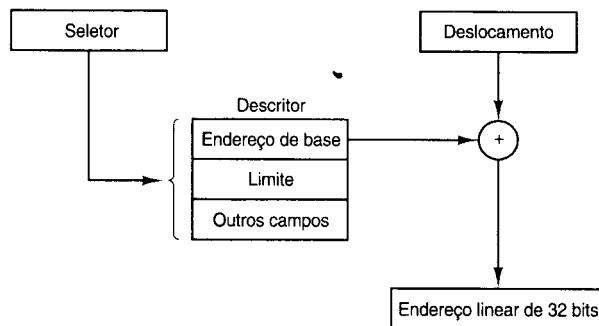



Figura 4-29 A conversão de um par (seletor, deslocamento) em um endereço linear.

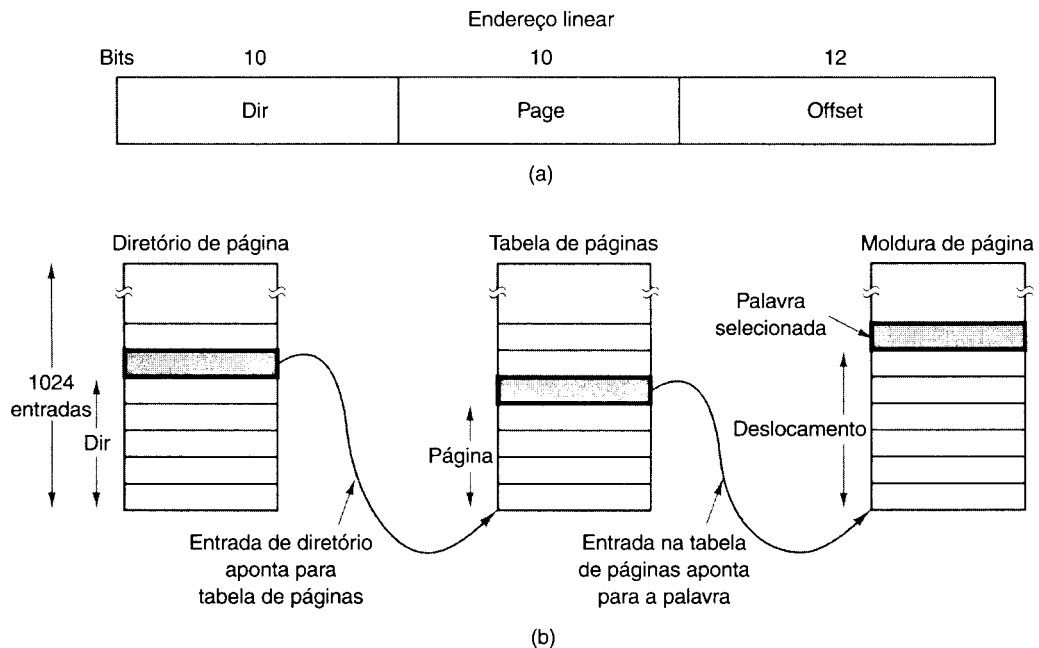



Figura 4-30 Mapeamento de um endereço linear para um endereço físico.

para incluir *Base* no final das contas é permitir segmentação pura (não-paginada) e por questão de compatibilidade com os 286, que têm a paginação sempre desativada (i. e., os 286 têm somente segmentação pura, mas não paginação).

Também vale notar que se algum aplicativo não necessita de segmentação, mas está contente com um único espaço de endereço de 32 bits paginado, esse modelo é possível. Todos os registradores de segmento podem ser configurados com o mesmo seletor, cujo descritor tem *Base* = 0 e *Limite* configurado como o máximo. O deslocamento da instrução, então, será o endereço linear, com somente um único espaço de endereço utilizado — com efeito, uma paginação normal.

Em suma, devemos congratular os projetistas do Pentium. Dado os objetivos contraditórios de implementar paginação pura, segmentação pura e segmentos paginados, e, ao mesmo tempo, ser compatível com os 286 e fazer tudo isso eficientemente, o projeto resultante é surpreendentemente simples e limpo.

Embora tenhamos abordado a arquitetura completa da memória virtual do Pentium, mesmo que resumidamente, vale dizer algumas palavras sobre a proteção, uma vez que esse assunto está intimamente relacionado com a memória virtual. Assim como o esquema de memória virtual é bastante semelhante ao modelado no MULTICS, o sistema de proteção também o é. O Pentium suporta quatro níveis de proteção, com o nível 0 sendo o mais privilegiado, e o nível 3 o menos. Esses são mostrados na Figura 4-31. Em cada instante, um programa em execução está em um certo nível, indicado por um campo de 2 bits em seu PSW. Cada segmento no sistema também tem um nível.

Contanto que um programa restrinja-se a utilizar segmentos no seu próprio nível, tudo funciona bem. As tentativas de acessar dados em um nível mais alto são permitidas. As tentativas de acessar dados em um nível mais baixo são ilegais e geram interrupções. As tentativas de chamar procedimentos em um nível diferente (acima ou abaixo) são permitidas, mas de uma maneira cuidadosamente controlada. Para fazer uma chamada internível, a instrução CALL deve conter um seletor em vez de um endereço. Esse seletor designa um descritor chamado **portão de chamada (*call gate*)**, o qual dá o endereço do procedimento a ser chamado. Assim, não é possível saltar no meio de um segmento de código arbitrário em um nível diferente. Somente pontos de entrada oficiais podem ser utilizados. Os conceitos de níveis de proteção e de portões de chamada foram desbravados no MULTICS, onde foram vistos como **anéis de proteção.**

Uma utilização típica para esse mecanismo é sugerida na Figura 4-31. No nível 0, localizamos o *kernel* do sistema operacional, que trata a E/S, o gerenciamento de memória e outras questões críticas. No nível 1, o manipulador de chamadas do sistema está presente. Os programas de usuário podem chamar procedimentos aqui para fazer as chamadas de sistema executarem, mas somente uma lista específica e protegida de procedimentos pode ser chamada. O nível 2 contém procedimentos de biblioteca, possivelmente compartilhados entre muitos programas em execução. Os programas de usuário podem chamar esses pro-

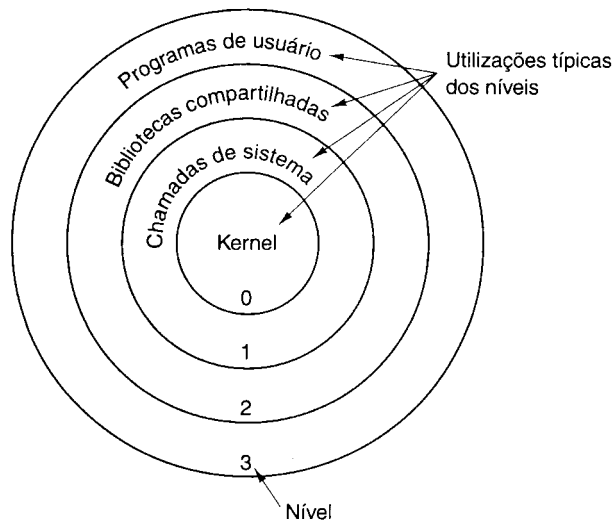



Figura 4-31 O esquema de proteção no Pentium.

cedimentos e ler seus dados, mas não podem modificá-los. Por fim, os programas de usuário executam no nível 3, que tem a menor proteção.

Interrupções utilizam um mecanismo semelhante aos portões de chamada. Elas também referenciam descritores, em vez de endereços absolutos, e esses descritores apontam para procedimentos específicos a serem executados. O campo *Type* na Figura 4-28 distingue entre segmentos de código, segmentos de dados e os vários tipos de portões.

4.7 VISÃO GERAL DO GERENCIAMENTO DE MEMÓRIA NO MINIX

O gerenciamento de memória no MINIX é simples: não se utiliza paginação nem troca. O gerenciador de memória mantém uma lista de lacunas classificadas pela ordem de endereço de memória. Quando memória é necessária, devido a uma chamada de sistema FORK ou EXEC, a lista de lacunas é pesquisada, utilizando o algoritmo do primeiro ajuste para uma lacuna que seja suficientemente grande. Uma vez que um processo foi colocado na memória, ele permanece exatamente no mesmo lugar até terminar. Ele nunca é enviado para disco e também nunca é movido para outro lugar na memória. Tampouco faz a área alocada crescer ou encolher.

Essa estratégia merece alguma explicação. Ela deriva de três fatores: (1) a idéia de que o MINIX destina-se a computadores pessoais e não a sistemas de compartilhamento de tempo de grande porte; (2) o desejo de ter o MINIX funcionando em todos os IBM PC e (3) o desejo de tornar simples e direta a implementação do sistema em outros computadores pequenos.

O primeiro fato significa que, na média, o número de processos em execução será pequeno, de modo que, em geral, haverá memória disponível suficiente para armazenar todos os processos com espaço de sobra. A troca em disco, então, não será exigida. Como acrescenta complexidade ao sistema, não fazer troca torna o código mais simples.

O desejo de ter o MINIX executando em todos os computadores compatíveis com IBM PC também teve um impacto significativo no projeto do gerenciamento de memória. Os sistemas mais simples nessa família utilizam o processador 8088, cuja arquitetura de gerenciamento de memória é muito primitiva. Ela não suporta memória virtual sob qualquer forma e nem mesmo detecta estouro de pilha, um defeito que tem implicações importantes sobre a maneira com que os processos são dispostos na memória. Tais limitações não existem na maioria dos projetos posteriores que utilizam os processadores 80386, 80486 ou Pentium. Entretanto, tirar proveito desses recursos tornaria o MINIX incompatível com muitas máquinas de categoria inferior que ainda são aproveitáveis e estão em uso.

A questão de portabilidade argumenta em favor do mais simples esquema de gerenciamento de memória possível. Se o MINIX utilizasse paginação ou segmentação, seria difícil, se não impossível, portá-lo para máquinas que não têm esses recursos. Fazendo o menor número possível de suposições quanto ao hardware, o número de máquinas para as quais o MINIX poder ser portado aumenta.

Outro aspecto incomum do MINIX é a maneira como o gerenciamento de memória é implementado. Ele não é parte do *kernel*. Em vez disso, ele é tratado pelo processo gerenciador de memória, que executa no espaço do usuário e comunica-se com o *kernel* pelo mecanismo padrão de mensagens. A posição do gerenciador de memória no nível de servidor é mostrada na Figura 2-26.

Mover o gerenciador de memória para fora do *kernel* é um exemplo da separação entre **política** e **mecanismo.** As decisões sobre quais processos serão colocados em quais lugares na memória (política) são feitas pelo gerenciador de memória. A configuração real dos mapas de memória para os processos (mecanismo) é feita pela tarefa de sistema dentro do *kernel*. Essa divisão torna relativamente fácil alterar a política de gerenciamento de memória (algoritmos, etc.) sem modificar as camadas mais baixas do sistema operacional.

A maior parte do código do gerenciador de memória é dedicada ao tratamento das chamadas de sistema do MINIX que envolvem gerenciamento de memória, principalmente FORK e EXEC, em vez de apenas manipular listas de processos e lacunas. Na próxima seção, veremos o *layout* da memória e, em seções posteriores, daremos uma passada de olhos em como as chamadas de sistema de gerenciamento de memória são processadas pelo gerenciador de memória.

4.7.1 Leiaute de Memória

Processos simples do MINIX utilizam espaços I e D combinados, em que todas as partes do processo (texto, dados e pilha) compartilham um bloco de memória que é alocado e liberado como um bloco. Os processos também podem ser compilados para utilizar espaços I e D separados. Para tornar o assunto mais claro, primeiro será discutida a alocação de memória para o modelo mais simples. Os processos que utilizam espaços I e D separados podem utilizar memória mais eficientemente, mas tirar proveito desse recurso complica as coisas. Discutiremos as complicações depois de delinear o caso mais simples.

A memória é alocada no MINIX em duas ocasiões. Em primeiro lugar, quando um processo cria um filho, a quantidade de memória necessária para o filho é alocada. Em segundo lugar, quando um processo altera sua imagem de memória via chamada de sistema EXEC, a imagem antiga é retornada à lista livre como uma lacuna, e memória é alocada para a nova imagem. A nova imagem pode estar em uma parte da memória diferente da memória liberada. Sua posição dependerá de onde uma lacuna adequada é encontrada. A memória também é liberada sempre que um processo termina, seja saindo normalmente ou eliminado por um sinal.

A Figura 4-32 mostra as duas maneiras de alocar memória. Na Figura 4-32(a), vemos dois processos, *A* e *B*, na memória. Se *A* cria um filho, obtemos a situação da Figura 43-2(b). O filho é uma cópia exata de *A*. Se o filho agora executar o arquivo *C*, a memória ficará parecida com a Figura 4-32(c). A imagem do filho é substituída por C.

Note que a memória antiga para o filho é liberada antes de a nova memória para *C* ser alocada, para que *C* possa utilizar a memória do filho. Dessa maneira, uma série de pares FORK e EXEC (tal como o *shell* configurando uma canalização) resulta em todos os processos serem adjacentes, sem lacunas entre eles, como seria o caso se a nova memória tivesse sido alocada antes de a memória antiga ter sido liberada.

Quando a memória é alocada, seja por FORK seja por EXEC, uma certa quantidade é tomada para o novo processo. No primeiro caso, a quantidade tomada é idêntica à que o processo do pai tem. No último caso, o gerenciador de memória toma a quantidade especificada no cabeçalho do arquivo executado. Uma vez que essa alocação foi feita, sob nenhuma condição o processo aloca mais memória.

O que foi dito até agora aplica-se a programas que foram compilados com os espaços I e D combinados. Os programas com espaços I e D separados tiram proveito de um modo expandido de gerenciamento de memória denominado **texto compartilhado.** Quando tal processo faz um FORK, somente a quantidade de memória necessária para uma cópia dos dados e da pilha do novo processo é alocada. Ambos, pai e filho, compartilham o código executável já em uso pelo pai. Quando tal processo faz um EXEC, é feita uma pesquisa na tabela de processos para ver se outro processo já está utilizando o código executável necessário. Se um for localizado, nova memória é alocada somente para

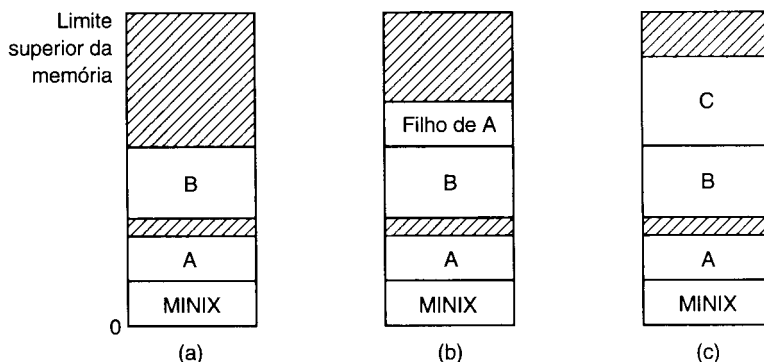



Figura 4-32 A alocação de memória. (a) Originalmente. (b) Depois de um FORK. (c) Depois que o filho fez um EXEC. As regiões sombreadas correspondem à memória livre. O processo é do tipo I&D comuns.

os dados e para a pilha, enquanto o texto já na memória é compartilhado. Texto compartilhado complica a terminação de um processo. Quando um processo termina, ele sempre libera a memória ocupada por seus dados e pela pilha. Mas ele somente libera a memória ocupada por seu segmento de texto depois que uma pesquisa na tabela de processos revela que nenhum outro processo está compartilhando essa memória. Assim, pode ser alocada mais memória a um processo quando ele inicia do que é liberada quando ele termina, se ele carregou o próprio texto quando iniciou, mas esse texto está sendo compartilhado por um ou mais outros processos quando o primeiro processo termina.

A Figura 4-33 mostra como um programa é armazenado na forma de um arquivo de disco e como isso é transferido para o arranjo interno de memória de um processo MINIX. O cabeçalho no arquivo de disco contém as informações sobre os tamanhos das diferentes partes da imagem, assim como o tamanho total. No cabeçalho de um programa com espaços I e D comuns, um campo especifica o tamanho total das partes de texto e de dados; essas partes são copiadas diretamente para a imagem da memória. A parte de dados na imagem é aumentada pela quantidade especificada no campo *bss* no cabeçalho. Essa área é limpa para conter apenas zeros e é utilizada para dados estáticos não-inicializados. A quantidade total de memória a ser alocada é especificada pelo campo *total* no cabeçalho. Se, por exemplo, um programa tem 4K de texto, 2K de dados mais *bss* e 1K de pilha, e o cabeçalho diz para alocar 40K no total, a lacuna de memória não-utilizada entre o segmento de dados e o segmento de pilha será de 33K. Um arquivo de programa no disco também pode conter uma tabela de símbolos. Esta última é para uso na depuração e não é copiada para a memória.

Se o programador souber que a memória total necessária para o crescimento dos segmentos de dados e de pilha combinados para o arquivo *a.out* é no máximo 10K, ele pode dar o comando

```
chmem =10240 a.out,
```

o que altera o campo de cabeçalho de modo que, no momento do EXEC, o gerenciador de memória aloca um espaço de 10240 bytes mais que a soma dos segmentos de texto e dados iniciais. Para o exemplo acima, um total de 16K será alocado em todos os subseqüentes EXECs do arquivo. Dessa quantidade, os 1K superiores serão utilizados para a pilha, e os 9K restantes estarão na lacuna, onde podem ser utilizados para o crescimento da pilha, da área de dados ou ambos.

Para um programa que utiliza espaços I e D separados (indicado por um bit no cabeçalho que é configurado pelo *linkeditor*), o campo total no cabeçalho aplica-se somente aos espaços de dados e de pilha combinados. Para um programa com 4K de texto, 2K de dados, 1K de pilha e um tamanho total de 64K serão alocados 68K (4K de espaço de instruções, 64K de espaço de dados), deixando 61K para o segmento de dados e a pilha consumirem durante a execução. O limite do segmento de dados pode ser movido somente pela chamada de sistema BRK. Tudo que a BRK faz é verificar se o novo segmento de dados colide com o ponteiro atual da pilha e, se não, anotar a alteração em algumas tabelas internas. Isso é inteiramente interno à memória originalmente alocada para o processo; nenhuma memória adicional é alocada pelo sistema operacional. Se o novo segmento de dados colide com a pilha, a chamada falha.

Essa estratégia foi escolhida para tornar possível executar o MINIX em um IBM PC com um processador 8088, que não verifica estouro de pilha em hardware. Um programa de usuário pode empurrar tantas palavras quanto quiser sobre a pilha sem que o sistema operacional esteja ciente disso. Em computadores com hardware de gerenciamento de memória mais sofisticado, é alocada uma certa quantidade de memória para a pilha inicialmente. Se esta última tentar crescer além dessa quantidade, ocorre uma interrupção para o sistema operacional, e o sistema aloca outro pedaço de memória para a pilha, se possível. Essa interrupção não existe nos 8088, tornando perigoso ter a pilha adjacente a qualquer coisa exceto um trecho grande de memória não-utilizada, uma vez que a pilha pode crescer rapidamente e sem aviso. O MINIX foi projetado de modo que quando é implementado em um computador com melhor gerenciamento de memória, é simples e direto alterar o gerenciador de memória do MINIX.

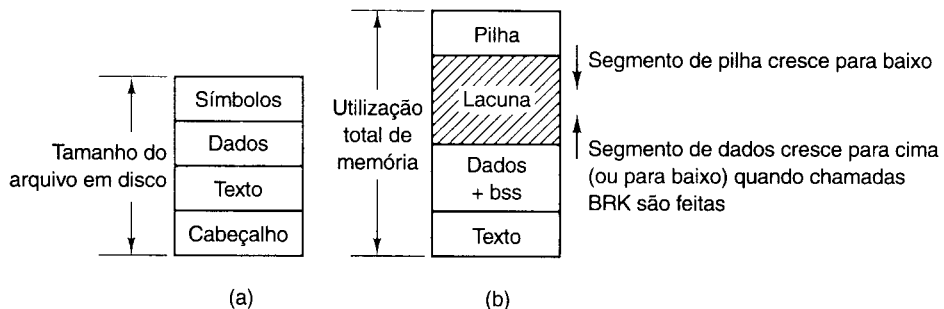



Figura 4-33 (a) Um programa como armazenado em um arquivo de disco. (b) Arranjo interno de memória para um único processo. Nas duas partes da figura, o disco ou o endereço de memória mais baixo está no fundo, e o endereço mais alto está no topo.

Este é um bom lugar para mencionar uma possível dificuldade semântica. Quando utilizamos a palavra "segmento", referimo-nos a uma área da memória definida pelo sistema operacional. Os processadores Intel 80x86 têm um conjunto interno de "registradores de segmento" e (nos processadores mais avançados) "tabelas de descritores de segmentos" que oferecem suporte de hardware para "segmentos". O conceito de segmento dos projetistas de hardware da Intel é semelhante, mas não é sempre o mesmo que os segmentos utilizados e definidos pelo MINIX. Todas as referências a segmentos neste texto devem ser interpretadas como referências a áreas da memória delineadas pelas estruturas de dados do MINIX. Vamos nos referir explicitamente a registradores de segmento ou a descritores de segmento quando falarmos sobre o hardware.

Essa advertência pode ser generalizada. Os projetistas de hardware freqüentemente tentam proporcionar suporte ao sistema operacional que eles esperam ser utilizado em suas máquinas, e a terminologia utilizada para descrever registradores e outros aspectos de uma arquitetura de processador normalmente reflete uma idéia de como os recursos serão utilizados. Tais recursos são freqüentemente úteis para o implementador de um sistema operacional, mas eles podem não ser utilizados da mesma maneira que o projetista de hardware previu. Isso pode levar a mal-entendidos quando a mesma palavra tiver significados diferentes quando utilizada para descrever um aspecto de um sistema operacional ou do hardware subjacente.

4.7.2 Processamento de Mensagens

Como todos os outros componentes do MINIX, o gerenciador de memória é baseado em mensagens. Depois que o sistema foi iniciado, o gerenciador de memória entra no seu laço principal, que consiste em esperar uma mensagem, em executar a solicitação contida na mensagem e em enviar uma resposta. A Figura 4-34 fornece a lista de tipos de mensagem válidos, seus parâmetros de entrada e o valor enviado de volta na mensagem de resposta.

FORK, EXIT, WAIT, WAITPID, BRK e EXEC estão, claramente, intimamente relacionadas com alocação e com desalocação de memória. As chamadas KILL, ALARM e PAUSE são, todas relacionadas a sinais, assim como o são SIGACTION, SIGSUSPEND, SIGPENDING, SIGMASK e SIGRETURN. Estas últimas também podem afetar o que está na memória, porque, quando um sinal elimina um processo, a memória do processo é desalocada. REBOOT tem efeitos por todo o sistema operacional, mas seu primeiro trabalho é enviar sinais para terminar todos os processos de uma maneira controlada; assim, o gerenciador de memória é um bom lugar para ele. As sete chamadas GET/SET não têm absolutamente nada a ver com gerenciamento de memória. Elas também não têm nada a ver com o sistema de arquivos. Mas elas precisam entrar no sistema de arquivos ou no gerenciador de memória, uma vez que cada chamada de sistema é tratada por um ou por outro. Elas foram colocadas aqui simplesmente porque o sistema de arquivos já estava suficientemente grande. PTRACE, que é utilizada em depuração, está aqui pela mesma razão.

A mensagem final, KSIG, não é uma chamada de sistema. KSIG é o tipo de mensagem utilizado pelo *kernel* para informar o gerenciador de memória sobre um sinal que se origina no *kernel*, como SIGINT, SIGQUIT ou SIGALRM.

Embora haja uma rotina de biblioteca *sbrk*, não há nenhuma chamada de sistema SBRK. A rotina de biblioteca computa a quantidade de memória necessária, adicionando ao tamanho atual incremento ou decremento especificado como parâmetro e faz uma chamada BRK para configurar o tamanho. De maneira semelhante, não há chamadas de sistema separadas para *geteuid* e *getegid*. As chamadas GETUID e GETGID retornam ambos os identificadores reais e efetivos. De maneira semelhante, GETPID retorna o *pid* do processo de chamada e de seu pai.

Uma estrutura de dados-chave utilizada para processamento de mensagens é a tabela *call_vec* declarada em *table.c* (linha 16515). Ela contém ponteiros para os procedimentos que tratam os vários tipos de mensagens. Quando uma mensagem entra no gerenciador de memória, o laço principal extrai o tipo de mensagem e coloca na variável global *mm_call*. Esse valor é, então, utilizado como índice em *callvec* a fim de localizar o ponteiro para o procedimento que trata a mensagem recém-chegada. Esse procedimento, então, é chamado para executar a chamada de sistema. O valor que ele retorna é enviado de volta para o chamador na mensagem de resposta, a fim de informar sobre o sucesso ou sobre fracasso da chamada. Esse mecanismo é semelhante ao da Figura 1-16, só que no espaço do usuário em vez de no espaço do *kernel*.

4.7.3 Estruturas de Dados e Algoritmos do Gerenciador de Memória

O gerenciador de memória tem duas estruturas de dados-chave: a tabela de processos e a tabela de lacunas. Agora veremos cada uma delas individualmente.

Na Figura 2-4 vimos que alguns campos da tabela de processos são necessários para o gerenciamento de processo, outros para o gerenciamento de memória e outros ainda para o sistema de arquivos. No MINIX, cada uma dessas três partes do sistema operacional tem sua própria tabela de processos, contendo somente os campos de que ela necessita. Para simplificar as coisas, as entradas correspondem-se exatamente. Assim, a entrada *k* da tabela do gerenciador de memória refere-se ao mesmo processo que a entrada *k* da tabela do sistema de arquivos. Quando um processo é criado ou destruído, todas as três partes atualizam suas tabelas para refletir a nova situação, a fim de mantê-las sincronizadas.

A tabela de processos do gerenciador de memória é chamada *mproc;* sua definição está em */usr/src/mm/mproc.h*. Ela contém todos os campos relacionados com a alocação de memória de um processo, assim como alguns itens adicionais. O campo mais importante é a matriz *mp_seg*, que tem três entradas, para os segmentos de texto, dados e pi-

Tipo da mensagem	Parâmetros da entrada	Valor da resposta
FORK	(nenhum)	*Pid* do filho, (para filho: 0)
EXIT	Status da saída	(Nenhuma resposta se bem-sucedida)
WAIT	(nenhum)	Status
WAITPID	(nenhum)	Status
BRK	Novo tamanho	Novo tamanho
EXEC	Ponteiro para pilha inicial	(Nenhuma resposta se bem-sucedida)
KILL	Identificador do processo e do sinal	Status
ALARM	Número de segundos a esperar	Tempo residual
PAUSE	(nenhum)	(Nenhuma resposta se bem-sucedida)
SIGACTION	Número do sinal, ação, ação antiga	Status
SIGSUSPEND	Máscara de sinal	(Nenhuma resposta se bem-sucedida)
SIGPENDING	(nenhum)	Status
SIGMASK	Como configuração, configuração antiga	Status
SIGRETURN	Contexto	Status
GETUID	(nenhum)	*Uid, uid* efetivo
GETGID	(nenhum)	*Gid, gid* efetivo
GETPID	(nenhum)	*Pid, pid* do pai
SETUID	Novo *uid*	Status
SETGID	Novo *gid*	Status
SETSID	Novo *sid*	Grupo do processo
GETPGRP	Novo *gid*	Grupo do processo
PTRACE	Solicitação, *pid*, endereço, dados	Status
REBOOT	Como (suspende, reinicializa ou pane)	(Nenhuma resposta se bem-sucedida)
KSIG	Entrada de processos e de sinais	(Nenhuma resposta)

Figura 4-34 Os tipos de mensagem, de parâmetros de entrada e de valores de resposta utilizados para comunicar com o gerenciador de memória.

lha, respectivamente. Cada entrada é uma estrutura que contém o endereço virtual, o endereço físico e o comprimento do segmento, todos medidos em cliques em vez de em bytes. O tamanho de um clique é dependente da implementação; para o MINIX-padrão é 256 bytes. Todos os segmentos devem iniciar em um limite de clique e ocupar um número integral de cliques.

O método utilizado para registrar a alocação de memória é mostrado na Figura 4-35. Nessa figura, temos um processo com 3K de texto, 4K de dados, uma lacuna de 1K e, então, uma pilha de 2K, para uma alocação total de memória de 10K. Na Figura 4-35(b) vemos o que são os campos de comprimento, físico e virtual, para cada um dos três segmentos, supondo que o processo não tenha espaços I e D separados. Nesse modelo, o segmento de texto está sempre vazio, e o segmento de dados contém tanto texto quanto dados. Quando um processo referencia o endereço virtual 0, seja para saltar para ele ou para lê-lo (i. e., como espaço de instrução ou como espaço de dados), o endereço físico 0x32000 (em decimal, 200K) será utilizado. Esse endereço está no clique 0x320.

Note que o endereço virtual em que a pilha começa depende inicialmente da quantidade total de memória alocada para o processo. Se o comando *chmem* for utilizado para modificar o cabeçalho do arquivo a fim de proporcionar uma área de alocação dinâmica maior (uma lacuna maior entre os segmentos de dados e de pilha), da próxima vez que o arquivo for executado, a pilha iniciaria em um endereço virtual mais alto. Se a pilha crescer mais um clique, a entrada da pilha *deverá* mudar da tripla (0x20, 0x340, 0x8) para a tripla (1F 0x, 0x33F, 0x9).

O hardware 8088 não tem uma interrupção de limite de pilha, e o MINIX define a pilha de maneira que ela não desencadeie a interrupção em processadores de 32 bits até que a pilha já tenha sobrescrito o segmento de dados. Assim, essa alteração não será feita até a próxima chamada de sistema BRK, momento em que o sistema operacional explicitamente lê SP e recalcula as entradas de segmento.

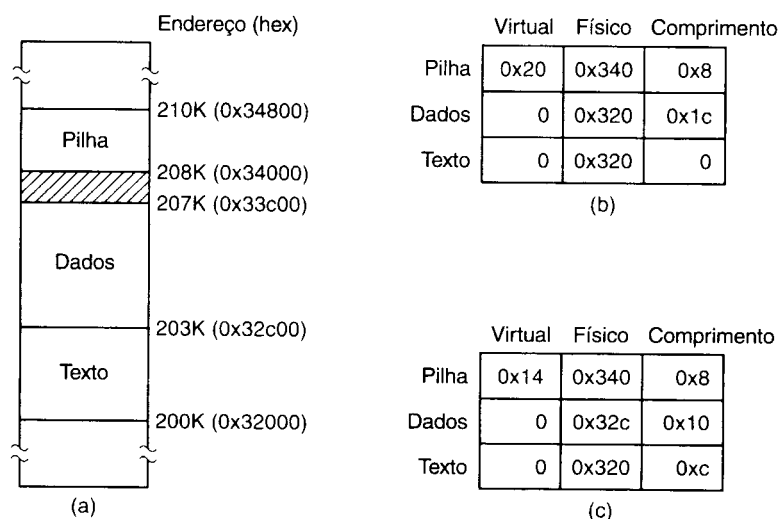



Figura 4-35 (a) Um processo na memória. (b) Sua representação de memória para os espaços I e D não-separados. (c) Sua representação de memória para os espaços I e D separados.

Em uma máquina com uma interrupção de pilha, a entrada do segmento de pilha poderia ser atualizada logo que a pilha ultrapassasse seu segmento. Isso não é feito pelo MINIX em processadores Intel de 32 bits, pelas razões que agora discutiremos.

Mencionamos anteriormente que os esforços dos projetistas de hardware nem sempre podem produzir exatamente o que o projetista de software necessita. Mesmo no modo protegido em um Pentium, o MINIX não interrompe quando a pilha supera seu segmento. Embora no modo protegido, o hardware Intel detecte acesso à memória tentado fora de um segmento (como definido por um descritor de segmento como o da Figura 4-28), no MINIX o descritor do segmento de dados e o descritor do segmento de pilha são sempre idênticos. Os dados e a pilha definidos pelo MINIX utilizam parte desse espaço e, portanto, qualquer um ou os dois pode expandir-se na lacuna entre eles. Entretanto, somente o MINIX pode gerenciar isso. A CPU não tem como detectar erros que envolvem a lacuna, uma vez que, no que diz respeito ao hardware, a lacuna é uma parte válida da área de dados e da área da pilha. Naturalmente, o hardware pode detectar um erro muito grande, tal como a tentativa de acessar memória a partir de fora da área combinada de dados, da lacuna e da pilha. Isso protegerá um processo de erros de outros processos, mas não é suficiente para proteger um processo de si mesmo.

Uma decisão de projeto foi feita aqui. Reconhecemos que pode ser feito um argumento em favor do abandono do segmento compartilhado definido pelo hardware, o que permite que o MINIX realoque dinamicamente a área da lacuna. A alternativa, utilizar o hardware para definir segmentos de pilha e de dados que não se sobrepõem, ofereceria alguma segurança adicional a certos erros, mas tornaria o MINIX mais faminto por memória. O código-fonte está disponível para qualquer pessoa que queira avaliar a outra abordagem.

A Figura 4-35(c) mostra as entradas de segmento para o arranjo de memória da Figura 4-35(a) para espaços I e D separados. Aqui, os dois segmentos, de texto e de dados, são diferentes de zero no comprimento. A matriz *np_seg* mostrada na Figura 4-35(b) ou (c) é principalmente utilizada para mapear endereços virtuais para endereços de memória físicos. Dado um endereço virtual e o espaço a que pertence, é uma questão simples ver se o endereço virtual é válido ou não (i. e., cai dentro de um segmento) e, se válido, qual é o endereço físico correspondente. O procedimento de *kernel umap* executa esse mapeamento para as tarefas de E/S e para copiar para e do espaço do usuário, por exemplo.

O conteúdo das áreas de dados e de pilha que pertence a um processo pode ser modificado enquanto o processo executa, mas o texto não muda. É comum para vários processos estar executando cópias do mesmo programa; por exemplo, vários usuários podem estar executando o mesmo *shell*. A eficiência da memória é melhorada, utilizando texto compartilhado. Quando está para carregar um processo, EXEC abre o arquivo que armazena a imagem de disco do programa a ser carregado e lê o cabeçalho de arquivo. Se o processo utiliza espaços I e D separados, é feita uma pesquisa nos campos *mp_dev*, *mp_ino* e *mp_ctime* em cada entrada de *mproc*. Essas armazenam os números de dispositivo e nó-i e o tempo/hora de alteração do status das imagens sendo executadas por outros processos. Se um processo já carregado é encontrado executando o mesmo programa que está para ser carregado, não há nenhuma necessidade de alocar memória para outra cópia do texto.

Em vez disso, a porção *mp_seg[T]* do mapa de memória do novo processo é inicializada para apontar para o mesmo lugar onde o segmento de texto já está carregado e somente os dados e de partes de pilha são configurados em uma nova alocação de memória. Isso é mostrado na Figura 4-36. Se o programa utiliza espaços I e D combinados ou nenhuma coincidência for localizada, a memória será alocada como mostrado na Figura 4-35, e o texto e os dados para o novo processo serão copiados do disco.

Além das informações de segmento, *mproc* também armazena o ID de processo (*pid*) do próprio processo e de seu pai, os *uids* e os *gids* (tanto reais como efetivos), as informações sobre sinais e o status de saída, se o processo já terminou, mas seu pai ainda não fez uma WAIT para ele.

A outra tabela importante do gerenciador de memória é a tabela de lacunas, *hole*, definida em *alloc.c*, que lista cada lacuna na memória pela ordem ascendente do endereço de memória. As lacunas entre os segmentos de dados e de pilha não são consideradas lacunas; elas já foram alocadas a processos. Portanto, elas não estão contidas na lista de lacunas livres. Cada entrada da lista de lacunas tem três campos: o endereço de base da lacuna, em cliques; o comprimento da lacuna, em cliques; e um ponteiro para a próxima entrada na lista. A lista é simplesmente encadeada, de modo que é fácil localizar o início da próxima lacuna a partir de qualquer dada lacuna, mas para localizar a lacuna anterior, você precisa pesquisar a lista inteira desde o início até que você chegue a uma dada lacuna.

A razão de registrar tudo sobre segmentos e lacunas em cliques em vez de bytes é simples: é muito mais eficiente. No modo de 16 bits, inteiros de 16 bits são utilizados para registrar endereços de memória; portanto, com cliques de 256 bits, até 16MB de memória podem ser suportados. No modo de 32 bits, campos de endereço podem referenciar até 2^{40} bytes, o que representa 1024 gigabytes.

As principais operações na lista de lacunas são a alocação de um pedaço de memória de um tamanho dado e retornar uma alocação existente. Para alocar memória, a lista de lacunas é pesquisada, começando na lacuna com o endereço mais baixo, até que uma lacuna grande o suficiente seja encontrada (primeiro ajuste). O segmento, então, é alocado, reduzindo-se a lacuna pela quantidade necessá-

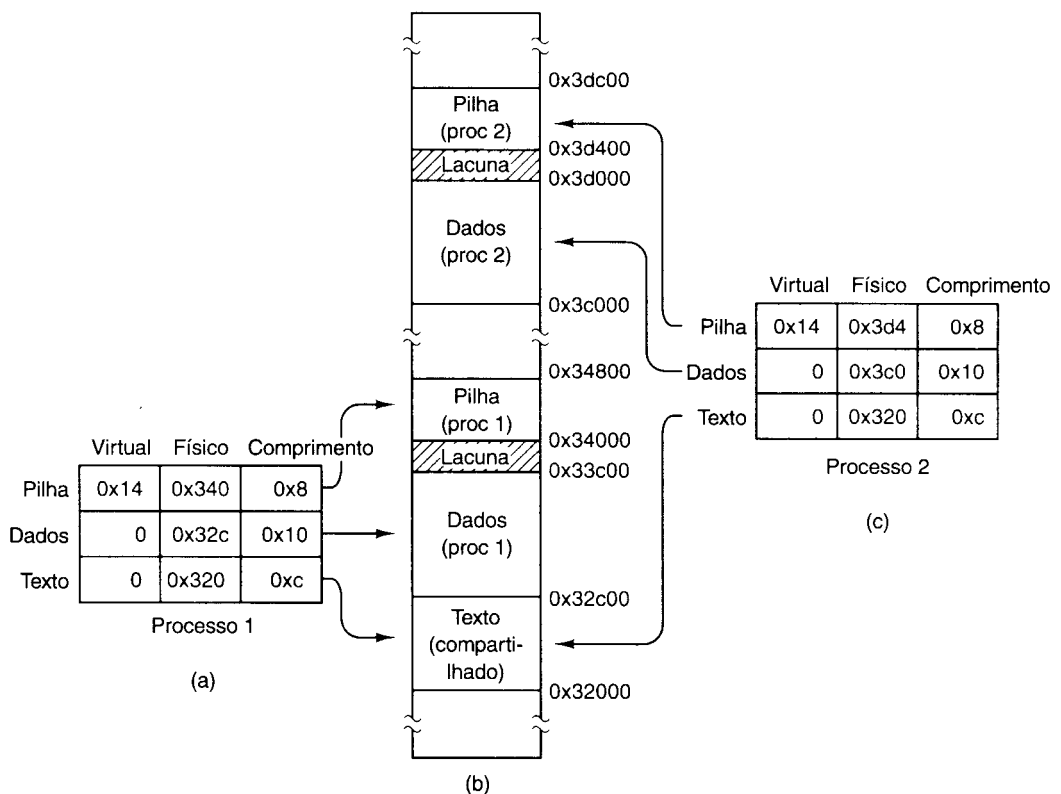



Figura 4-36 (a) O mapa de memória de um processo de espaços I e D separados, como na figura anterior. (b) O arranjo da memória depois que um segundo processo inicia, executando a mesma imagem do programa com texto compartilhado. (c) O mapa de memória do segundo processo.

ria para o segmento ou, no caso raro de um ajuste exato, removendo-se a lacuna da lista. Esse esquema é rápido e simples, mas padece tanto de uma pequena quantidade de fragmentação interna (até 255 bytes podem ser desperdiçados no clique final, uma vez que sempre é tomado um número integral de cliques) como de fragmentação externa.

Quando um processo termina e é limpo, sua memória de pilha e de dados é retornada para a lista de livres. Se utiliza I e D comuns, isso libera toda sua memória, uma vez que tais programas nunca têm uma alocação separada de memória para texto. Se o programa utiliza I e D separados, e uma consulta na tabela de processos revela que nenhum outro processo está compartilhando o texto, a alocação de texto também será retornada. Uma vez que com texto compartilhado as regiões de texto e de dados não são necessariamente contíguas, duas regiões da memória podem ser retornadas. Para cada região retornada, se um dos ou os dois vizinhos da região forem lacunas, eles são fundidos, portanto lacunas adjacentes nunca ocorrem. Dessa maneira, o número, a posição e o tamanho das lacunas variam continuamente durante a operação do sistema. Sempre que todos os processos de usuário terminam, toda a memória disponível fica mais uma vez pronta para alocação. Mas isso não é necessariamente uma única lacuna, uma vez que a memória física pode ser interrompida por regiões não-utilizáveis pelo sistema operacional, como nos sistemas compatíveis com IBM nos quais a memória ROM e a memória reservada para transferências de E/S separam a memória usável abaixo do endereço 640K, da memória acima de 1M.

4.7.4 Chamadas de Sistema FORK, EXIT e WAIT

Quando processos são criados ou destruídos, a memória deve ser alocada ou desalocada. Além disso, a tabela de processos deve ser atualizada, incluindo as partes mantidas pelo *kernel* e o FS. O gerenciador de memória coordena toda essa atividade. A criação de processo é feita pela chamada FORK e realizada na série de passos mostrada na Figura 4-37.

É difícil e inconveniente parar uma chamada FORK no meio do caminho, assim o gerenciador de memória mantém sempre uma contagem do número de processos atualmente em existência para ver facilmente se uma entrada da tabela de processos está disponível. Se a tabela não estiver cheia, é feita uma tentativa de alocar memória para o filho. Se o programa é um programa com espaços I e D separados, é necessária memória suficiente apenas para novas alocações de dados e de pilha. Se esse passo também tiver sucesso, seguramente FORK funcionará. A memória recentemente alocada é, então, preenchida, uma entrada de processo é localizada e preenchida, um *pid* é escolhido e as outras partes do sistema são informadas de que um novo processo foi criado.

Um processo termina completamente quando dois eventos aconteceram juntos: (1) o próprio processo saiu (ou foi eliminado por um sinal) e (2) seu pai executou uma chamada de sistema WAIT para saber o que aconteceu. Um processo que saiu ou foi eliminado, mas cujo pai não fez (ainda) uma WAIT para ele, entra em um tipo de animação suspensa, às vezes, conhecido como **estado zumbi.** Ele é impedido de ser agendado e tem seu temporizador de alarme desligado (se estiver ligado), mas não é removido da tabela de processos. Sua memória é liberada. O estado zumbi é temporário e raramente dura muito. Quando o pai, por fim, faz a WAIT, a entrada da tabela de processos é liberada e o sistema de arquivos e o *kernel* são informados.

Um problema surge se o próprio pai de um processo que está saindo já estiver morto. Se nenhuma ação especial for tomada, o processo que está saindo permaneceria um zumbi eternamente. Em vez disso, as tabelas são alteradas para torná-lo um filho do processo *init*. Quando o sistema surge, *init* lê o arquivo */etc/ttytab* para obter uma lista de todos os terminais e, então, cria um processo de *login* para tratar cada uma. Então, ele bloqueia, esperando os processos terminarem. Dessa maneira, órfãos zumbis são removidos rapidamente.

1. Verificar se a tabela de processos está cheia.
2. Tentar alocar memória para os dados e para a pilha do filho.
3. Copiar os dados e a pilha do pai para a memória do filho.
4. Localizar uma entrada de processo livre e copiá-la do pai para ele.
5. Inserir o mapa de memória do filho na tabela de processos.
6. Escolher um *pid* para o filho.
7. Informar o *kernel* e o sistema de arquivos sobre filho.
8. Informar o mapa de memória do filho para o *kernel.*
9. Enviar mensagens de resposta para pai e para filho

Figura 4-37 Os passos exigidos para executar a chamada de sistema FORK.

4.7.5 Chamada de Sistema EXEC

Quando um comando é digitado no terminal, o *shell* cria um novo processo, que, então, executa o comando solicitado. Teria sido possível ter uma única chamada de sistema fazendo ambas, FORK e EXEC de uma vez, mas elas foram oferecidas como duas chamadas distintas por uma razão muito boa: facilitar a implementação de redirecionamento de E/S. Quando o *shell* cria um filho, se a entrada-padrão estiver redirecionada, o filho fecha a entrada-padrão e, então, abre a nova entrada-padrão antes de executar o comando. Dessa maneira, o processo recentemente iniciado herda a entrada-padrão redirecionada. A saída-padrão é tratada da mesma maneira.

EXEC é a chamada de sistema mais complexa no MINIX. Ela deve substituir a imagem atual da memória por uma nova, incluindo configurar uma nova pilha. Ela realiza seu trabalho em uma série de passos, como mostrado na Figura 4-38.

Cada passo consiste, por sua vez, de outros ainda menores, alguns dos quais podem falhar. Por exemplo, pode não haver memória suficiente. A ordem em que os testes são feitos foi cuidadosamente escolhida para certificar-se de que a imagem da memória antiga não foi liberada até que seja certo que o EXEC terá sucesso, para evitar a situação embaraçosa de não ser capaz de configurar uma nova imagem da memória, e também não ter a antiga para restaurar. Normalmente EXEC não retorna, mas se falhar, o processo de chamada deve obter o controle novamente, com uma indicação do erro.

Há alguns passos na Figura 4-38 que merecem mais comentários. Primeiro é a pergunta de se há ou não espaço suficiente. Depois de determinar quanta memória é necessária, o que exige determinar se a memória de texto de outro processo pode ser compartilhada, a lista de lacunas inteira é pesquisada para verificar se há memória física suficiente *antes* de liberar a memória antiga — se a memória antiga fosse liberada primeiro e houvesse memória insuficiente, seria difícil obter de volta a imagem antiga novamente.

Entretanto, esse teste é excessivamente estrito. Ele, às vezes, rejeita chamadas EXEC que, de fato, poderiam ter sucesso. Suponha, por exemplo, que o processo que faz a chamada EXEC ocupe 20K e seu texto não seja compartilhado por qualquer outro processo. Suponha ainda que haja uma lacuna de 30K disponível e que a nova imagem exija 50K. Testando antes de liberar, descobriremos que somente 30K estão disponíveis e rejeitaremos a chamada. Se tivéssemos liberado primeiro, talvez pudéssemos ter sucesso, dependendo de a nova lacuna de 20K ser ou não adjacente, portanto agora fundida com a lacuna de 30K. Uma implementação mais sofisticada poderia tratar essa situação um pouco melhor.

Outro possível aprimoramento seria pesquisar duas lacunas, uma para o segmento de texto e uma para o segmento de dados, se o processo a ser EXECutado utilizasse espaços I e D separados. Não há nenhuma necessidade de os segmentos serem contíguos.

Uma questão mais sutil é o arquivo executável ajustar-se no espaço de endereço virtual. O problema é que a memória não é alocada em bytes, mas em cliques de 256 bytes. Cada clique deve pertencer a um único segmento e não pode ser, por exemplo, metade dados, metade pilha, porque a administração inteira da memória está em cliques.

Para ver como essa restrição pode causar problemas, note que o espaço de endereço em sistemas de 16 bits (8088 e 80286) é limitado a 64K, o que pode ser dividido em 256 cliques. Suponha que um programa de espaços I e D separados tenha 40.000 bytes de texto, 32.770 bytes de dados e 32.760 bytes de pilha. O segmento de dados ocupa 129 cliques, dos quais o último é apenas parcialmente utilizado; entretanto, o clique inteiro é parte do segmento de dados. O segmento da pilha é 128 cliques. Juntos, eles excedem 256 cliques e, portanto, não podem coexistir, mesmo que o número de bytes necessários (mal) se ajuste no espaço de endereço virtual. Na teoria, esse problema existe em todas as máquinas cujo tamanho do clique é maior que 1 byte, mas, na prática, raramente ocorre em processadores da classe Pentium, uma vez que esses permitem grandes segmentos (4 GB).

1. Verificar permissões — o arquivo é executável?
2. Ler o cabeçalho para obter os tamanhos dos segmentos e o tamanho total.
3. Buscar os argumentos e o ambiente do chamador.
4. Alocar nova memória e liberar memória antiga não-necessária.
5. Copiar a pilha para a nova imagem da memória.
6. Copiar o segmento de dados (e possivelmente de texto) para a nova imagem da memória.
7. Verificar e tratar os bits *setuid, setgid*.
8. Corrigir entradas da tabela de processos.
9. Informar o *kernel* que o processo agora é executável.

Figura 4-38 Os passos exigidos para executar a chamada de sistema EXEC.

Outra questão importante é como a pilha inicial é configurada. A chamada de biblioteca normalmente utilizada para invocar EXEC com argumentos e com um ambiente é

```
execve(name, argv, envp);
```

onde *name* é um ponteiro para o nome do arquivo a ser executado, *argv* é um ponteiro para uma matriz de ponteiros, cada um apontando para um argumento, e *envp* é um ponteiro para uma matriz de ponteiros, cada um apontando para uma *string* do ambiente.

Seria muito fácil implementar EXEC colocando simplesmente os três ponteiros na mensagem para o gerenciador de memória e deixá-lo buscar o nome do arquivo e as duas matrizes sozinhas. Então, ele teria de buscar cada argumento e cada *string* uma por vez. Fazer isso dessa maneira exige pelo menos uma mensagem à tarefa de sistema por argumento ou por *string*, e provavelmente mais, uma vez que o gerenciador de memória não tem como saber o tamanho de cada um de antemão.

Para evitar o *overhead* de múltiplas mensagens lendo todos esses pedaços, uma estratégia completamente diferente foi escolhida. O procedimento de biblioteca *execve* constrói a pilha inicial inteira dentro de si mesmo e passa seu endereço de base e seu tamanho para o gerenciador de memória. Criar a nova pilha dentro do espaço usuário é bastante eficiente, porque as referências a argumentos e a *strings* são referências de memória locais, não referências a um espaço de endereço diferente.

Para tornar esse mecanismo mais claro, considere um exemplo. Quando um usuário digita

```
ls –l f.c g.c
```

para o *shell*, o *shell* interpreta-o e, então, faz a chamada

```
execve("/bin/ls", argv, envp);
```

para o procedimento de biblioteca. O conteúdo das duas matrizes de ponteiros é mostrado na Figura 4-39(a). O procedimento *execve*, dentro do espaço de endereço do *shell*, agora constrói a pilha inicial, como mostrado na Figura 4-39(b). Essa pilha por fim é copiada intacta para o gerenciador de memória durante o processamento da chamada EXEC.

Quando a pilha por fim é copiada para o processo do usuário, ela não será colocada no endereço virtual 0. Em vez disso, ela será colocada no fim da memória alocada, como determinado pelo campo total de tamanho de memória no cabeçalho do arquivo executável. Como um exemplo, vamos arbitrariamente supor que o tamanho total é 8192 bytes, então, o último byte disponível para o programa está no endereço 8191. Cabe ao gerenciador de memó-

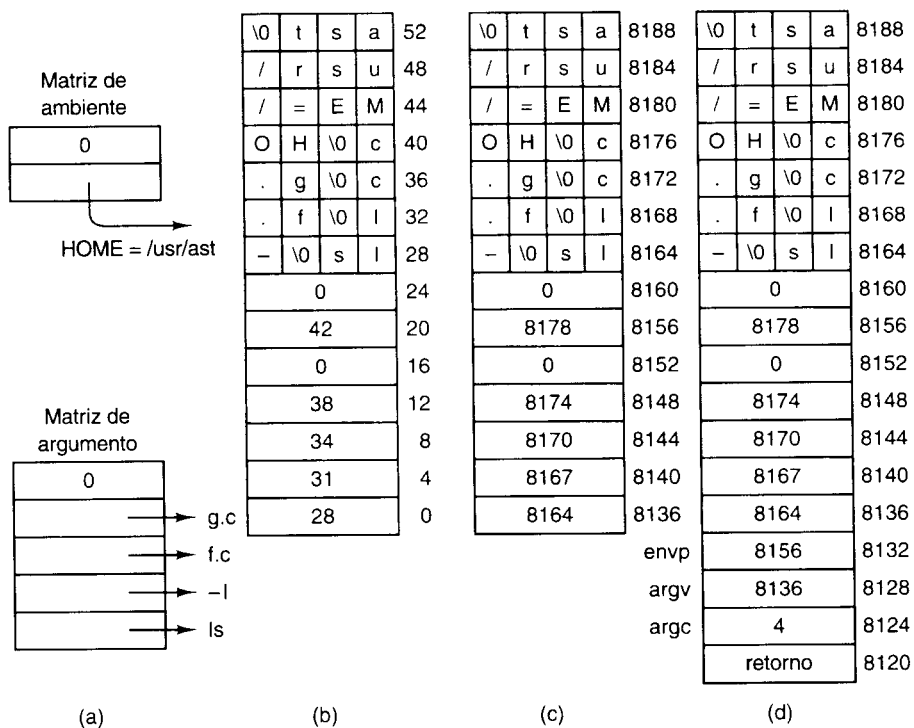



Figura 4-39 (a) As matrizes passadas a *execve*. (b) A pilha construída por *execve*. (c) A pilha após a realocação pelo gerenciador de memória. (d) A pilha como aparece para *main* no início da execução.

ria realocar os ponteiros dentro da pilha de modo que quando depositado no novo endereço a pilha fique parecida com a Figura 4-39(c)

Quando a chamada EXEC é concluída, e o programa começa a executar, a pilha, de fato, ficará exatamente como na Figura 4-39(c), com o ponteiro da pilha tendo o valor 8136. Entretanto, outro problema ainda precisa ser resolvido. O programa principal do arquivo executado provavelmente declarou algo assim:

```
main(argc, argv, envp);
```

no que diz respeito ao compilador de C, *main* é somente outra função. Não sabe que *main* é especial, então, compila o código para acessar os três parâmetros na suposição de que eles serão passados de acordo com a convenção-padrão de chamada de C, último parâmetro primeiro. Com um inteiro e dois ponteiros, espera-se os três parâmetros ocuparem as três palavras exatamente antes do endereço de retorno. Naturalmente, a pilha da Figura 4-39(c) não se parece com isso de modo algum.

A solução é fazer com que os programas não comecem com *main*. Em vez disso, uma pequena rotina de linguagem *assembly*, *crtso*, o procedimento *start-off* de tempo de execução do C, sempre é vinculada no endereço 0 do texto para que obtenha controle primeiro. Seu trabalho é empurrar mais três palavras sobre a pilha e, então, chamar *main*, utilizando a instrução-padrão de chamada. Isso resulta na pilha da Figura 4-39(d) no momento em que *main* começa a executar. Assim, *main* é enganada, pensando que foi chamada da maneira normal (na realidade, não é um truque de verdade; ela *é* chamada dessa maneira).

Se o programador omite a chamada a *exit* no fim de *main*, o controle passará de volta para a rotina de *start-off* quando *main* termina. Novamente, o compilador apenas vê *main* como um procedimento comum e gera o código normal para retornar dele depois da última declaração. Assim, *main* retorna para seu chamador, a rotina de *start-off* que, então, chama *exit*. A maior parte do código de 32 bits de *crtso* é mostrada na Figura 4-40. Os comentários devem esclarecer sua operação. Tudo que foi deixado de fora é o código que carrega os registradores que são colocados na pilha, e algumas linhas que configuram um sinalizador, indicando se um co-processador de ponto flutuante está presente ou não.

4.7.6 Chamada de Sistema de BRK

Os procedimentos de biblioteca *brk* e *sbrk* são utilizados para ajustar o limite superior do segmento de dados. O primeiro toma um tamanho absoluto (em bytes) e chama BRK. O último toma um incremento positivo ou negativo para o tamanho atual, calcula o novo tamanho de segmento de dados e, então, chama BRK. Não há uma chamada de sistema SBRK real.

Uma pergunta interessante é: "Como *sbrk* monitora o tamanho atual, de modo que possa calcular o novo tamanho?" A resposta é que uma variável, *brksize*, sempre armazena o tamanho atual de modo que *sbrk* possa localizá-lo. Essa variável é inicializada para um símbolo gerado pelo compilador que fornece o tamanho inicial de texto mais dados (I e D não-separados) ou dados apenas (I e D separados). O nome e, de fato, a própria existência desse símbolo depende do compilador e, portanto, ele não será encontrado definido em qualquer arquivo de cabeçalho nos diretórios dos arquivos-fonte. Ele é definido na biblioteca, no arquivo *brksize.s*. O lugar exato onde ele estará localizado depende do sistema, mas ele estará no mesmo diretório que *crtso.s*.

Executar BRK é fácil para o gerenciador de memória. Tudo que deve ser feito é verificar se tudo ainda cabe no espaço de endereço, ajustar as tabelas e informar o *kernel*.

4.7.7 Manipulação de Sinais

No Capítulo 1, os sinais foram descritos como um mecanismo para transportar as informações para um processo que não necessariamente está esperando entrada. Há um conjunto definido de sinais e cada um deles tem uma ação-padrão — seja eliminar um processo para o qual é dirigido, seja ignorar o sinal. O processamento de sinais seria fácil de entender e de implementar se estas fossem as únicas alternativas. Entretanto, processos podem utilizar chamadas de sistema que alteram essas respostas. Um processo pode solicitar que qualquer sinal (exceto para o sinal especial SIGKILL) seja ignorado. Ademais, um processo pode preparar para **capturar** um sinal, solicitando que um procedimento de **manipulação de sinal** interno ao processo seja ativado em vez da ação padrão para qualquer sinal (exceto, novamente, para SIGKILL). Portanto, ao progra-

```
push   ecx          ! empilha environ
push   edx          ! empilha argv
push   eax          ! empilha argc
call   _main        ! main(argc, argv, envp)
push   eax          ! empilha o status de saída
call   _exit
hlt                 ! força uma interrupção se a saída falhar
```

Figura 4-40 A parte-chave da rotina *start-off* de tempo de execução do C.

mador aparece que há dois tempos distintos quando o sistema operacional lida com sinais: uma fase de preparação quando um processo pode modificar sua resposta para um sinal futuro, e para uma fase de resposta quando um sinal é gerado e sofre uma ação. A ação pode ser a execução de um manipulador de sinais personalizado. Na verdade, há uma terceira fase. Quando um manipulador escrito pelo usuário termina, uma chamada de sistema especial limpa e restaura a operação normal do processo sinalizado. O programador não precisa tomar conhecimento sobre essa terceira fase. Ele escreve um manipulador de sinal como qualquer outra função. O sistema operacional cuida dos detalhes de invocar e de terminar o manipulador e de gerenciar a pilha.

Na fase de preparação, há várias chamadas de sistema que um processo pode executar em qualquer momento para alterar sua resposta para um sinal. A mais geral dessas é SIGACTION, que pode especificar que o processo ignore alguns sinais, capture algum sinal (substituindo a ação-padrão pela execução de um código de tratamento de sinal definido pelo usuário dentro do processo) ou restaure a resposta-padrão para algum sinal. Outra chamada de sistema, SIGPROCMASK, pode bloquear um sinal, fazendo com que ele seja enfileirado e seja executado somente quando e se o processo desbloquear aquele sinal particular em um momento mais tarde. Essas chamadas podem ser feitas a qualquer momento, mesmo dentro de uma função de captura de sinal. No MINIX, a fase de preparação de processamento de sinal é tratada inteiramente pelo gerenciador de memória, uma vez que as estruturas de dados necessárias estão todas na parte do gerenciador de memória da tabela de processos. Para cada processo, há diversas variáveis *sigset_t*, nas quais cada possível sinal é representado por um bit. Uma variável desse tipo define um conjunto de sinais que devem ser ignorados, outra define um conjunto que deve ser capturado e assim por diante. Para cada processo, também há uma matriz de estruturas *sigaction*, uma para cada sinal. Cada elemento da estrutura *sigaction* contém uma variável para armazenar o endereço de um manipulador personalizado para esse sinal, e uma variável *sigset_t* adicional para mapear sinais a serem bloqueados enquanto aquele manipulador está executando. O campo utilizado para o endereço do manipulador pode, em vez disso, armazenar valores especiais que significam que o sinal deve ser ignorado ou deve ser tratado na maneira padrão definida para esse sinal.

Quando um sinal é gerado, múltiplas partes do sistema MINIX podem ser envolvidas. A resposta começa no gerenciador de memória, o qual descobre quais processos devem obter o sinal, utilizando as estruturas de dados recém-mencionadas. Se o sinal deve ser capturado, ele deve ser entregue para o processo de destino. Isso requer salvar as informações sobre o estado do processo, de tal modo que a execução normal possa ser reassumida. As informações são armazenadas na pilha do processo sinalizado, e uma verificação deve ser feita para determinar se há espaço de pilha suficiente. O gerenciador de memória faz essa verificação, uma vez que isso está dentro do seu âmbito e, então, chama a tarefa de sistema no *kernel* para colocar as informações na pilha. A tarefa de sistema também manipula o contador de programa do processo, então o processo pode executar o código do manipulador. Quando o manipulador termina, uma chamada de sistema SIGRETURN é feita. Por essa chamada, tanto o gerenciador de memória como o *kernel* participam da restauração do contexto e dos registradores do processo para que a execução normal possa ser reassumida. Se o sinal não for capturado, a ação-padrão é tomada, o que pode envolver chamar o sistema de arquivos para produzir um ***dump* de núcleo** (gravando a imagem do processo em um arquivo que pode ser examinado com um depurador), assim como eliminar o processo, o que envolve o sistema de arquivos, o gerenciador de memória e o *kernel*. Por fim, o gerenciador de memória pode dirigir uma ou mais repetições dessa ação, uma vez que um único sinal pode precisar ser entregue para um grupo de processos.

Os sinais conhecidos para o MINIX são definidos em */usr/include/signal.h*, um arquivo exigido pelo padrão POSIX. Eles são listados na Figura 4-41. Todos os sinais exigidos pelo POSIX são definidos no MINIX, mas nem todos eles são atualmente suportados. Por exemplo, o POSIX exige vários sinais relacionados com controle de *jobs*, com a capacidade de colocar em segundo plano um programa em execução e de trazê-lo de volta. O MINIX não suporta controle de *jobs*, mas programas que talvez gerem esses sinais podem ser portados para o MINIX. Se gerados, tais sinais serão ignorados. O MINIX também define alguns sinais não-POSIX e alguns sinônimos para nomes POSIX para compatibilidade com código-fonte mais antigo.

Os sinais podem ser gerados de duas maneiras: pela chamada de sistema KILL e pelo *kernel*. Os sinais gerados pelo *kernel* do MINIX sempre incluem SIGINT, SIGQUIT e SIGALRM. Outros sinais do *kernel* dependem de suporte de hardware. Por exemplo, os processadores 8086 e 8088 não suportam detecção de códigos ilegais de operação de instrução, mas essa capacidade está disponível nos 286 e nos superiores, que interrompem uma tentativa de executar um *opcode* ilegal. Esse serviço é oferecido pelo hardware. O implementador do sistema operacional deve oferecer código para gerar um sinal em resposta à interrupção. Vimos, no Capítulo 2, que *kernel/exception.c* contém código para fazer exatamente isso para diversas condições diferentes. Assim, um sinal SIGILL pode ser gerado em resposta a uma instrução ilegal quando o MINIX executa em processador 286 ou superior, mas esse sinal nunca será visto quando o MINIX executar em um 8088.

Apenas porque o hardware pode gerar uma interrupção em uma certa condição, não significa que essa capacidade possa ser utilizada completamente pelo implementador do sistema operacional. Por exemplo, vários tipos de violações de integridade de memória resultam em exceções em todos os processadores Intel, começando com os 286. O código em *kernel/exception.c* traduz essas exceções em sinais SIGSEGV. Há exceções separadas geradas para infrações dos limites do segmento de pilha definido pelo hardware e para outros segmentos, uma vez que estes talvez precisam ser

Sinal	Descrição	Gerado por
SIGHUP	Desconecta (*hangup*)	Chamada de sistema KILL
SIGINT	Interrupção	*Kernel*
SIGQUIT	Encerrar	*Kernel*
SIGILL	Instrução ilegal	*Kernel* (*)
SIGTRAP	Interrupção de depuração	*Kernel* (M)
SIGABRT	Terminação anormal	*Kernel*
SIGFPE	Exceção de ponto flutuante	*Kernel* (*)
SIGKILL	Eliminar (não pode ser capturado nem pode ser ignorado)	Chamada de sistema KILL
SIGUSR1	Sinal definido pelo usuário 1	Não suportado
SIGSEGV	Violação de segmentação	*Kernel* (*)
SIGUSR2	Sinal definido pelo usuário 2	Não suportado
SIGPIPE	Gravação em um pipe sem ninguém para lê-lo	*Kernel*
SIGALRM	Alarme do relógio, tempo limite	*Kernel*
SIGTERM	Sinal de software para encerramento proveniente de kill	Chamada de sistema KILL
SIGCHLD	Processo-filho terminado ou parado	Não-suportado
SIGCONT	Continua se parado	Não-suportado
SIGSTOP	Sinal de parada	Não-suportado
SIGTSTP	Sinal de parada interativo	Não-suportado
SIGTTIN	Processo em segundo plano quer ler	Não-suportado
SIGTTOU	Processo em segundo plano quer escrever	Não-suportado

Figura 4-41 Sinais definidos pelo POSIX e pelo MINIX. Sinais indicados por (*) dependem de suporte de hardware. Sinais marcados com (M) não são definidos pelo POSIX, mas o são pelo MINIX para compatibilidade com programas mais antigos. Vários nomes obsoletos e sinônimos não estão listados aqui.

tratados de maneira diferente. Entretanto, por causa da maneira como o MINIX utiliza a memória, o hardware não pode detectar todos os erros que talvez ocorram. O hardware define uma base e um limite para cada segmento. A base do segmento de dados definida pelo hardware é a mesma que a base do segmento de dados do MINIX, mas o limite definido pelo hardware do segmento de dados é mais alto do que o limite que o MINIX impõe em software. Em outras palavras, o hardware define o segmento de dados como a maior quantidade de memória que o MINIX possivelmente poderia utilizar para dados, se por alguma razão a pilha reduzir-se a nada. De maneira similar, o hardware define a pilha como a quantidade máxima de memória que a pilha do MINIX poderia utilizar se a área de dados pudesse reduzir-se a nada. Embora certas infrações possam ser detectadas pelo hardware, este não pode detectar a infração mais provável de pilha, o crescimento da pilha na área de dados, uma vez que no que diz respeito aos registradores de hardware e às tabelas de descritores a área de dados e a área da pilha sobrepõem-se.

Concebivelmente algum código poderia ser adicionado ao *kernel*, que verificaria os registradores de cada processo depois de cada vez que o processo obtivesse uma chance de executar e geraria um sinal SIGSEGV ao detectar uma infração da integridade das áreas de dados e de pilha definidas no MINIX. O benefício de fazer isso é incerto; as interrupções de hardware podem capturar uma violação imediatamente. Uma verificação de software poderia não ter uma chance de fazer seu trabalho até que muitas milhares de instruções adicionais tivessem sido executadas e, nesse ponto, talvez haja muito pouco que um manipulador de sinal possa fazer para tentar recuperar.

Qualquer que seja sua a origem, o gerenciador de memória processa todos os sinais da mesma maneira. Para cada processo a ser sinalizado, diversas verificações são feitas para ver se o sinal é praticável. Um processo pode sinalizar outro se o sinalizador é o superusuário ou se o *uid* real ou efetivo do sinalizador é igual ao *uid* real ou efetivo do processo sinalizado. Mas há várias condições que podem impedir que um sinal seja enviado. Zumbis não podem ser sinalizados, por exemplo. Um processo não pode ser sinalizado se ele tiver explicitamente chamado SIGACTION para ignorar o sinal ou SIGPROCMASK para bloqueá-lo. Bloquear um sinal é diferente de ignorá-lo; a recepção de um sinal bloqueado é memorizada e é entregue quando e se o processo sinalizado remover o bloqueio. Por fim, se

seu espaço de pilha não for adequado, o processo sinalizado é eliminado.

Se todas as condições são satisfeitas, o sinal pode ser enviado. Se o processo não arranjou para o sinal ser capturado, nenhuma informação precisa ser passada para o processo. Nesse caso o gerenciador de memória executa a ação-padrão para o sinal, que é normalmente eliminar o processo, possivelmente produzindo também um *dump* de núcleo. Para alguns sinais, a ação-padrão é ignorar o sinal. O POSIX exige que os sinais marcados como "Não-suportado" na Figura 4-41 sejam definidos, mas eles são ignorados pelo MINIX.

Capturar um sinal significa executar o código personalizado de tratamento de sinal do processo, cujo endereço é armazenado em uma estrutura *sigaction* na tabela de processos. No Capítulo 2, vimos como a moldura de pilha de um processo dentro de sua entrada na tabela de processos recebe as informações necessárias para reiniciar o processo quando ele é interrompido. Modificando-se a moldura de pilha de um processo a ser sinalizado, pode-se arranjar as coisas para que, quando o processo em seguida tenha permissão para executar, o manipulador de sinal execute. Modificando a própria pilha do processo no espaço do usuário, pode-se arranjar as coisas para que, quando o manipulador de sinal termine, a chamada de sistema SIGRETURN seja feita. Essa chamada de sistema nunca é invocada por código escrito pelo usuário, ela é executada depois que o *kernel* coloca seu endereço na pilha de tal maneira que seu endereço torna-se o endereço de retorno retirado da pilha, quando um manipulador de sinal termina. SIGRETURN restaura a moldura de pilha original do processo sinalizado, para que ele possa reassumir a execução no ponto onde foi interrompido pelo sinal.

Embora a etapa final de enviar um sinal seja feita pela a tarefa de sistema, esse é um bom lugar para resumir-se como ela é feita, uma vez que os dados utilizados são passados para o *kernel* pelo gerenciador de memória. Capturar um sinal requer algo muito parecido com a comutação de contexto que ocorre quando um processo é tirado da execução e outro processo é colocado em execução, uma vez que quando o manipulador conclui, o processo deve ser capaz de continuar como se nada tivesse acontecido. Entretanto, há somente um lugar na tabela de processos para armazenar o conteúdo de todos os registradores da CPU que são necessários para restaurar o processo ao seu estado original. A solução para esse problema é mostrada na Figura 4-42. A parte (a) da figura é uma visão simplificada da pilha de um processo e parte de sua entrada na tabela de processos logo depois que ele foi tirado da execução após uma interrupção. No momento da suspensão, o conteúdo de todos os registradores da CPU é copiado para a estrutura da moldura de pilha na entrada da tabela de processos desse processo, na parte do *kernel* da tabela de processos. Essa será a situação no momento em que um sinal for gerado, uma vez que um sinal é gerado por um processo ou por uma tarefa diferente do destinatário pretendido.

Na preparação da manipulação dos sinais, a moldura de pilha da tabela de processos é copiada para a própria pilha do processo como uma estrutura *sigcontext*, assim preservando-a. Então, uma estrutura *sigframe* é colocada na pilha. Essa estrutura contém as informações a serem utilizadas por SIGRETURN depois que o manipulador é concluído. Ela também contém o endereço do procedimento de biblioteca que invoca a própria SIGRETURN, *ret addr1* e outros endereços de retorno, *ret addr2*, que é o endereço onde a execução do programa interrompido será reassumida. Como será visto, entretanto, o último endereço não é utilizado durante a execução normal.

Embora o manipulador seja escrito como um procedimento costumeiro pelo programador, ele não é chamado por uma instrução de chamada. O campo do ponteiro de instrução (contador de programa) na moldura de pilha na tabela de processos é alterado para fazer o manipulador de sinal começar a executar quando *restart* coloca o processo sinalizado de volta em execução. A Figura 4-42(b) mostra a situação depois que essa preparação foi completada e como o manipulador de sinal executa. Lembre-se de que o manipulador de sinal é um procedimento usual, então, quando ele termina, *ret addr1* é retirado da pilha e SIGRETURN executa.

A parte (c) mostra a situação enquanto SIGRETURN está executando. O restante da estrutura *sigframe* são agora variáveis locais de SIGRETURN. Parte da ação de SIGRETURN é ajustar seu próprio ponteiro de pilha de tal modo que se ela fosse uma função para terminar como uma função comum, ela utilizaria *ret addr2* como seu endereço de retorno. Entretanto, SIGRETURN realmente não termina dessa maneira. Ela termina como outras chamadas de sistema, permitindo que o agendador no *kernel* decida qual processo reiniciar. Por fim, o processo sinalizado será reagendado e reiniciará nesse endereço, porque o endereço também está na moldura de pilha do processo original. A razão pela qual esse endereço está na pilha é que o usuário talvez queira rastrear um programa, utilizando um depurador e isso engana o depurador, fazendo-o ter uma interpretação razoável da pilha, enquanto um manipulador de sinal está sendo rastreado. Em cada fase, a pilha parece-se com a de um processo comum, com variáveis locais sobre um endereço de retorno.

O trabalho real de SIGRETURN é restaurar as coisas para o estado em que elas estavam antes de o sinal ter sido recebido, além da limpeza. Sobretudo, a moldura de pilha na tabela de processos é restaurada ao seu estado original, utilizando a cópia que foi salva na pilha do processo sinalizado. Quando SIGRETURN termina, a situação será como a da Figura 4-42(d), que mostra o processo que espera, voltando à execução no mesmo estado em que estava quando foi interrompido.

Para a maioria dos sinais, a ação-padrão é eliminar o processo sinalizado. O gerenciador de memória cuida disso para qualquer sinal que não seja ignorado por padrão e cujo processo destinatário não foi habilitado para tratar,

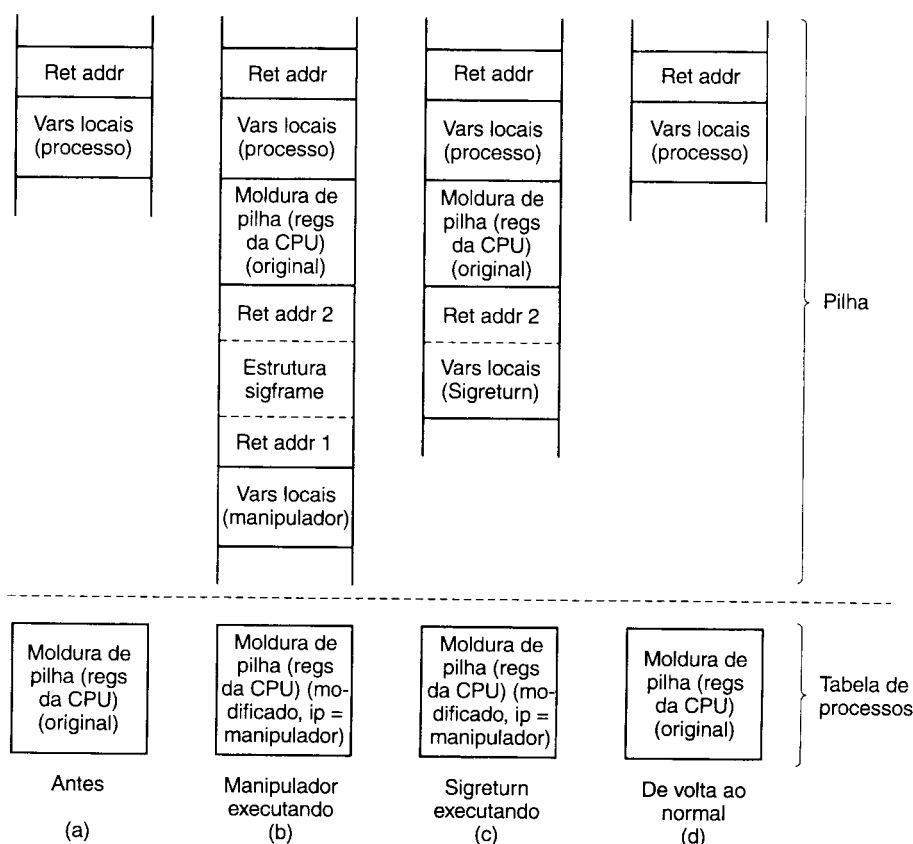



Figura 4-42 Uma pilha de processo (acima) e sua moldura de pilha na tabela de processos (embaixo), correspondendo a fases no tratamento de um sinal. (a) Estado enquanto o processo é tirado de execução. (b) Estado quando o manipulador começa a executar. (c) Estado quando SIGRETURN está executando. (d) Estado depois que SIGRETURN completa a execução.

bloquear ou para ignorar. Se o pai está esperando-o, o processo eliminado é limpo e removido da tabela de processos. Se o pai não o está esperando, ele se torna um zumbi. Para certos sinais (p. ex., SIGQUIT), o gerenciador de memória também grava um *dump* de núcleo do processo no diretório atual.

Facilmente pode acontecer de um sinal ser enviado para um processo que atualmente está bloqueado, esperando por um READ em um terminal para o qual nenhuma entrada está disponível. Se o processo não especificou que o sinal deve ser capturado, ele é simplesmente eliminado da maneira normal. Se, entretanto, o sinal é para ser capturado, surge a questão sobre o que fazer depois que a interrupção de sinal foi processada. O processo deve voltar ao estado de espera ou deve continuar com a próxima declaração?

O que o MINIX faz é isto: a chamada de sistema é terminada de tal maneira que ela retorna o código de erro *EINTR* para que o processo possa ver que a chamada foi interrompida por um sinal. Determinar se um processo sinalizado foi bloqueado em uma chamada de sistema não é inteiramente trivial. O gerenciador de memória deve solicitar que o sistema de arquivos verifique por ele.

Esse comportamento é sugerido, mas não é exigido, pelo POSIX, que também permite que um READ retorne o número de bytes lidos até o momento da recepção do sinal. Retornar *EINTR* torna possível configurar um alarme e capturar SIGALRM. Essa é uma maneira fácil de implementar um limite de tempo, por exemplo, terminando *login* e desligando uma linha de modem se um usuário não responder dentro de um certo período de tempo. A tarefa de relógio síncrono pode ser utilizada para fazer a mesma coisa com menor *overhead*, mas é uma invenção do MINIX e não tão portável quanto a utilização de sinais. Além disso, está disponível somente para processos de servidor e não para processos de usuário comuns.

4.7.8 Outras Chamadas de Sistema

O gerenciador de memória trata algumas chamadas de sistema mais simples. As funções de biblioteca *getuid* e *ge-*

teuid invocam a chamada de sistema GETUID, que retorna os dois valores em sua mensagem de retorno. De maneira semelhante, a chamada de sistema GETGID também retorna valores efetivos reais para serem utilizados pelas funções *getgid* e *getegid*. GETPID funciona da mesma maneira para retornar o ID do processo; e o ID do processo do pai, e SETUID e SETGID podem configurar os valores tanto reais como efetivos em uma chamada. Há duas chamadas de sistema adicionais nesse grupo, GETPGRP e SETSID. A primeira retorna o ID de grupo do processo, e a última configura-o como o valor do *pid* atual. Essas sete são as chamadas de sistema mais simples do MINIX.

As chamadas de sistema PTRACE e REBOOT também são tratadas pelo gerenciador de memória. A primeira suporta depuração de programas. A última afeta muitos aspectos do sistema. Ela é apropriada para ser colocada no gerenciador de memória porque sua primeira ação é enviar sinais para eliminar todos os processos, exceto *init*. Depois disso, ela chama o sistema de arquivos e a tarefa de sistema para completarem seu trabalho.

4.8 IMPLEMENTAÇÃO DO GERENCIAMENTO DE MEMÓRIA NO MINIX

Munidos de uma visão geral de como o gerenciador de memória funciona, voltemos agora a examinar o código em si. O gerenciador de memória é escrito inteiramente em C, sendo simples e direto e contém uma quantidade substancial de comentários no próprio código, de modo que nossa abordagem da maioria das partes não precisa ser longa ou complicada. Veremos primeiro um resumo dos arquivos de cabeçalho, depois o programa principal e, por fim, os arquivos para os vários grupos de chamadas de sistema discutidos anteriormente.

4.8.1 Arquivos de Cabeçalho e Estruturas de Dados

Vários arquivos de cabeçalho no diretório de fontes do gerenciador de memória têm os mesmos nomes que os arquivos no diretório do *kernel*, e esses nomes serão vistos novamente no sistema de arquivos. Tais arquivos têm funções semelhantes nos seus respectivos contextos. A estrutura paralela é projetada para facilitar o entendimento da organização global do sistema MINIX. O gerenciador de memória também tem diversos cabeçalhos com nomes únicos. Como em outras partes do sistema, o armazenamento para variáveis globais é reservado para quando é compilada a versão de *table.c* do gerenciador de memória. Nesta seção, veremos todos os arquivos de cabeçalho, assim como *table.c*.

Como com outras partes importantes do MINIX, o gerenciador de memória tem um arquivo de cabeçalho principal, o *mm.h* (linha 15800). Ele é incluído em cada compilação e, ele, por sua vez, inclui todos os grandes arquivos de cabeçalho do sistema de */usr/include* e seus subdiretórios que são necessários para cada módulo objeto. A maioria dos arquivos incluídos em *kernel/kernel. h* também é incluída aqui. O gerenciador de memória igualmente precisa de definições contidas em *include/fcntl.h* e *include/unistd.h*. As versões próprias do gerenciador de memória para *const.h*, *type.h*, *proto.h* e *glo.h*, também são incluídas.

Const.h (linha 15900) define algumas constantes utilizadas pelo gerenciador de memória, especialmente quando compilado para máquinas de 16 bits. A linha

```
#define printf printk
```

está aqui contida para que chamadas a *printf* sejam compiladas como chamadas à função *printk*. A função é semelhante àquela que vimos no *kernel* e é definida por uma razão semelhante, para que o gerenciador de memória possa exibir mensagens de erro e de depuração sem chamar o sistema de arquivos para ajudar.

Type.h atualmente não é utilizada e existe na forma de esqueleto apenas para que os arquivos do gerenciador de memória tenham a mesma organização que as outras partes do MINIX. *Proto.h* (linha 16100) reúne em um lugar protótipos de função necessários em todo o gerenciador de memória.

As variáveis globais do gerenciador de memória são declaradas em *glo.h* (linha 16200). O mesmo truque utilizado no *kernel* com *EXTERN* é aqui utilizado. Ou seja, *EXTERN* é normalmente uma macro que se expande para *extern*, exceto no arquivo *table.c*. Aí, ela se torna uma *string* nula para que se possa reservar espaço para as variáveis declaradas como *EXTERN*.

A primeira dessas variáveis, *mp*, é um ponteiro para uma estrutura *mproc*, a parte do gerenciador de memória na tabela de processos para o processo cuja chamada de sistema está sendo processada. A segunda variável *dont_reply* é iniciada como *FALSE* quando cada nova solicitação chega, mas pode ser configurada como *TRUE* durante a chamada, se for descoberto que nenhuma mensagem de resposta deve ser enviada. Por exemplo, nenhuma resposta é enviada para uma EXEC bem-sucedida. A terceira variável *procs_in_use*, monitora quantas entradas de processo estão atualmente em uso, tornando fácil ver se uma chamada FORK é praticável.

Os buffers de mensagens *mm_in* e *mm_out* são para as mensagens de solicitação e de resposta respectivamente. *Who* é o índice do processo atual; é relacionado a *mp* por

```
mp = &mproc [who].
```

Quando uma mensagem chega, o número da chamada de sistema é extraído dela e colocado em *mm_call*.

As três variáveis *err_code*, *result2* e *res_ptr* são utilizadas para armazenar valores retornados para o processo chamador na mensagem de resposta. A mais importante dessas variáveis é *err_code*, configurada como *OK* se a chamada for completada sem erro. As últimas duas variáveis são utilizadas quando um problema ocorre. O MINIX grava

uma imagem de um processo em um arquivo de núcleo quando um processo termina anormalmente. *Core_name* define o nome que o arquivo terá e *core_sset* é um mapa de bits que define quais sinais devem produzir *dumps* de núcleo.

A parte do gerenciador de memória na tabela de processos está no próximo arquivo, *mproc.h* (linha 16300). A maioria dos campos é adequadamente descrita por seus comentários. Vários campos lidam com tratamento de sinais. *Mp_ignore*, *mp_catch*, *mp_sigmask*, *mp_sigmask2* e *mp_sigpending* são mapas de bits, nos quais cada bit representa um dos sinais que pode ser enviado para um processo. O tipo *sigset_t* é um inteiro de 32 bits, portanto, o MINIX poderia com facilidade suportar até 32 sinais, mas atualmente apenas 16 sinais são definidos, com o sinal 1 sendo o bit menos significativo (mais à direita). Em qualquer caso, o POSIX exige funções padrão para adicionar ou para excluir membros dos conjuntos de sinais representados por esses mapas de bits para que toda manipulação necessária possa ser feita sem que o programador esteja ciente desses detalhes. A matriz *mp_sigact* é importante para tratar sinais. Há um elemento para cada tipo de sinal e cada elemento é uma estrutura *sigaction* (definida em *include/signal.h*). Cada estrutura *sigaction* consiste em três campos:

1. O campo *sa_handler* define se o sinal deve ser tratado na maneira padrão, ignorado, ou tratado por um manipulador especial.
2. O campo *sa_mask* é um *sigset_t* que define quais sinais devem ser bloqueados quando o sinal está sendo tratado por um manipulador personalizado.
3. O campo *sa_flags* é um conjunto de sinalizadores que se aplicam ao sinal.

Essa matriz possibilita grande flexibilidade no tratamento de sinais.

O campo *mp_flags* é utilizado para armazenar uma variada coleção de bits como indicado no fim do arquivo. Esse campo é um inteiro sem sinal de 16 bits em CPUs de categoria inferior, ou de 32 bits em um 386 e superiores. Há abundância de espaço para expansão aqui mesmo em 8088, uma vez que são utilizados apenas 9 bits.

O último campo na tabela de processos é *mp_procargs*. Quando um novo processo é iniciado, uma pilha como a mostrada na Figura 4-39 é construída, e um ponteiro para o início da matriz *argv* do novo processo é armazenado aqui. Este é utilizado pelo comando *ps*. Assim, para o exemplo da Figura 4-39, o valor 8164 seria armazenado aqui para permitir que *ps* exiba a linha de comando

```
ls –l f.c g.c
```

se executado enquanto o comando *ls* estiver ativo.

O próximo arquivo é *param.h* (linha 16400) que contém macros para muitos dos parâmetros de chamada de sistemas contidos na mensagem de solicitação. Ele também contém quatro macros para campos na mensagem de resposta. Quando a declaração

```
k = pid;
```

aparece em qualquer arquivo em que *param.h* é incluído, o pré-processador converte-o para

```
k = mm_in.m1_i1;
```

antes de utilizá-lo para alimentar o próprio compilador.

Antes de continuarmos com o código executável, vamos examinar *table.c* (linha 16500). Sua compilação reserva espaço de armazenamento para diversas variáveis e estruturas *EXTERN* que vimos em *glo.h* e *mproc.h*. A declaração

```
#define _TABLE
```

faz com que *EXTERN* torne-se uma *string* nula. Esse é o mesmo mecanismo que vimos no código do *kernel*.

O outro recurso importante de *table.c* é a matriz *call_vec* (linha 16515). Quando uma mensagem de solicitação chega, o número da chamada de sistema é extraído dela e utilizado como um índice em *call_vec* para localizar o procedimento que executa essa chamada de sistema. Todos os números de chamada de sistema que não correspondem a chamadas válidas invocam *no_sys*, que simplesmente retorna um código de erro. Note que, embora a macro *_PROTOTYPE* seja utilizada ao definir *call_vec*, isso não é uma declaração de um protótipo; é a definição de uma matriz inicializada. Entretanto, é uma matriz de funções, e a utilização de *_PROTOTYPE* é a maneira mais fácil de tornar isso compatível tanto com o C clássico (Kernighan & Ritchie) como com o C padrão.

4.8.2 Programa Principal

O gerenciador de memória é compilado e linkeditado independentemente do *kernel* e do sistema de arquivos. Conseqüentemente, ele tem seu próprio programa principal, iniciado depois que o *kernel* terminou sua própria inicialização. O programa principal está em *main.c*, na linha 16627. Depois de fazer sua própria inicialização chamando *mm_init*, o gerenciador de memória entra no seu laço na linha 16636, onde ele chama *get_work* para esperar uma mensagem de solicitação. Então, ele chama um dos procedimentos *do_XXX* via tabela *call_vec* para executar a solicitação e, por fim, enviar uma resposta, se necessário. Essa construção deve ser familiar para você agora: ela é a mesma utilizada pelas tarefas de E/S.

Os procedimentos *get_work* (linha 16663) e *reply* (linha 16676) tratam a recepção e o envio reais, respectivamente.

O último procedimento nesse arquivo é *mm_init*, que inicializa o gerenciador de memória. Ele não é utilizado depois que o sistema começa a executar. A chamada a *sys_getmap* na linha 16730 obtém as informações sobre a utilização de memória do *kernel*. O laço nas linhas 16734

a 16741 inicializa todas as entradas da tabela de processos para tarefas e para servidores, e as linhas seguintes preparam a entrada da tabela de processos referente a *init*. Na linha 16749, o gerenciador de memória espera o sistema de arquivos enviar-lhe uma mensagem. Como mencionado na discussão sobre tratamento de impasses no MINIX, essa é a única vez que o sistema de arquivos envia uma mensagem de solicitação para o sistema de arquivos. A mensagem informa quanta memória está sendo utilizada pelo disco de RAM. A chamada a *mem_init* na linha 16755 inicializa a lista de lacunas, chamando a tarefa de sistema. Depois disso, o gerenciamento de memória normal pode começar. Essa chamada também preenche as variáveis *total_clicks* e *free_clicks* que completam as informações que *mm_init* precisa para imprimir uma mensagem que mostra a memória total, a utilização da memória pelo *kernel*, o tamanho do disco de RAM e a memória livre. Depois da mensagem ser impressa, uma resposta é enviada para o sistema de arquivos (linha 16764), permitindo que ele continue. Por fim, à tarefa de memória é dado o endereço da parte do gerenciador de memória na tabela de processos para o benefício do comando *ps*.

4.8.3 Implementação de FORK, EXIT e WAIT

As chamadas de sistema FORK, EXIT e WAIT são implementadas pelos procedimentos *do_fork*, *do_mm_exit* e *do_wait* no arquivo *forkexit.c*. O procedimento *do_fork* (linha 16832) segue os passos mostrados na Figura 4-37. Note que a segunda chamada a *procs_in_use* (linha 16847) reserva as últimas entradas da tabela de processos ao superusuário. Ao calcular quanta memória o filho necessita, a lacuna entre os segmentos de dados e de pilha é incluída, mas o segmento de texto não o é. Tanto se o texto do pai é compartilhado ou, se o processo tiver espaços I e D comuns, seu segmento de texto é de comprimento zero. Depois de fazer a computação é feita uma chamada a *alloc_mem* para obter a memória. Se isso for bem-sucedido, os endereços de base do filho e do pai são convertidos de cliques em bytes absolutos e *sys_copy* é chamado para enviar uma mensagem à tarefa de sistema para realizar a operação de cópia.

Agora uma entrada está localizada na tabela de processos. O teste anterior envolvendo *procs_in_use* garante que uma existirá. Depois que a entrada foi localizada, ela é preenchida, primeiro copiando a entrada do pai aí e, então, atualizando os campos *mp_parent*, *mp_flags*, *mp_seg*, *mp_exitstatus* e *mp_sigstatus*. Alguns desses campos exigem tratamento especial. O bit *TRACED* no campo *mp_flags* é zerado, uma vez que um filho não herda o status de depuração. O campo *mp_seg* é uma matriz contendo elementos para os segmentos de dados de texto e de pilha, e a parte de texto é mantida, apontando para o segmento de texto do pai se os sinalizadores indicam que esse é um programa com I e D separados que pode compartilhar texto.

O próximo passo é atribuir um *pid* ao filho. A variável *next_pid* monitora o próximo *pid* a ser atribuído. Entretanto, é concebível que o seguinte problema poderia ocorrer. Depois de atribuir, digamos, *pid* 20 a um processo de vida muito longa, 30.000 outros processos poderiam ser criados e destruídos e *next_pid* poderia voltar a 20 novamente. Atribuir um *pid* que ainda está em uso seria um desastre (supondo que alguém mais tarde tentasse sinalizar o processo 20), então, pesquisamos a tabela inteira de processos para assegurar que o *pid* a ser atribuído já não está em uso.

As chamadas a *sys_fork* e a *tell_fs* informam o *kernel* e o sistema de arquivos respectivamente de que um novo processo foi criado para que possam atualizar suas tabelas de processos. (Todos os procedimentos que começam com *sys_* são rotinas de biblioteca que enviam uma mensagem à tarefa de sistema no *kernel*, solicitando um dos serviços da Figura 3-50). A criação e a destruição de processos sempre são iniciadas pelo gerenciador de memória e, então, propagadas para o *kernel* e para o sistema de arquivos quando concluídas.

A mensagem de resposta ao filho é enviada explicitamente no fim de *do_fork*. A resposta para o pai, contendo o *pid* do filho, é enviada pelo laço em *main*, como a resposta normal para uma solicitação.

A próxima chamada de sistema tratada pelo gerenciador de memória é EXIT. O procedimento *do_mm_exit* (linha 16912) aceita a chamada, mas a maior parte do trabalho é feita pela chamada para *mm_exit* algumas linhas mais para baixo. A razão para essa divisão de trabalho é que *mm_exit* também é chamada para cuidar de processos encerrados por um sinal. O trabalho é o mesmo, mas os parâmetros são diferentes, então, é conveniente dividir as coisas dessa maneira.

A primeira coisa que *mm_exit* faz é parar o temporizador se o processo tiver um executando. Em seguida, o *kernel* e o sistema de arquivos são notificados de que o processo não é mais executável (linhas 16949 e 16950). A chamada para o procedimento de biblioteca *sys_xit* envia uma mensagem à tarefa de sistema, instruindo-a a marcar o processo como não mais executável, assim ele não será mais agendado. Em seguida, a memória é liberada. Uma chamada a *find_share* determina se o segmento de texto está sendo compartilhado por outro processo e, se não, o segmento de texto é liberado por uma chamada a *free_mem*. Isso é seguido por outra chamada ao mesmo procedimento para liberar os dados e a pilha. Não vale a pena o problema de decidir se toda a memória poderia ser liberada em uma chamada a *free_mem*. Se o pai estiver esperando, *cleanup* é chamado para liberar a entrada na tabela de processos. Se o pai não estiver esperando, o processo torna-se um zumbi, indicado pelo bit *HANGING* na palavra *mp_flags*. Tanto no caso de o processo ter sido completamente eliminado como no de ele haver sido transformado em um zumbi, a ação final de *mm_exit* é fazer um laço pela tabela de processos e pesquisar os filhos do processo

que acaba de terminar (linhas 16975 a 16982). Se qualquer um for encontrado, ele é deserdado e torna-se filho de *init*. Se *init* estiver esperando, e um filho estiver pendurado, *cleanup*, então, é chamada para esse filho. Essa lida com situações como a mostrada na Figura 4-43(a), onde vemos que o processo 12 está para sair e que seu pai, 7, está esperando-o. *Cleanup* será chamada para livrar-se de 12, então, 52 e 53 transformam-se em filhos de *init*, como mostrado na Figura 4-43(b). Agora temos a situação de que 53, que já saiu, é o filho de um processo fazendo um WAIT. Conseqüentemente, ele também pode ser limpo.

Quando o processo pai faz um WAIT ou um WAITPID, o controle vai para o procedimento *do_waitpid* na linha 16992. Os parâmetros fornecidos pelas duas chamadas são diferentes, e as ações esperadas também, mas a configuração feita nas linhas 17009 a 17011 prepara variáveis internas para que *do_waitpid* possa executar as ações de qualquer das duas chamadas. O laço nas linhas 17019 a 17041 varre a tabela de processos inteira para ver se o processo tem algum filho e, se tiver, ele verifica se algum deles é um zumbi que agora pode ser limpo. Se um zumbi for encontrado (linha 17026), ele será limpo e *do_waitpid* retornará. O sinalizador *dont_reply* é ativado porque a resposta para o pai é enviada de dentro de *cleanup*, não do laço em *main*. Se um filho rastreado for encontrado, uma resposta será enviada para indicar que o processo está parado, e *do_waitpid* retorna. *Dont_reply* também é configurado como *true* para impedir que uma segunda resposta seja enviada por *main*.

Se o processo que faz o WAIT não tiver nenhum filho, ele simplesmente obterá o retorno de um erro (linha 17053). Se tiver filhos, mas nenhum for zumbi ou estiver sendo rastreado, será feito um teste para ver se *do_waitpid* foi chamada com um bit ligado para indicar que o pai não quer esperar. Se não (o caso normal), um bit é ligado na linha 17047 para indicar que ele está esperando, e o pai é suspenso até que um filho termine.

Quando um processo saiu e seu pai o está esperando, em qualquer ordem em que tais eventos ocorram, o procedimento *cleanup* (linha 17061) é chamado para executar os últimos rituais. Não há muito o que fazer a essa altura. O pai é acordado a partir de sua chamada WAIT ou WAITPID e recebe o *pid* do filho terminado, assim como os status de sinal e de saída dele. O sistema de arquivos já liberou a memória do filho, e o *kernel* já suspendeu o agendamento. Então, tudo o que o *kernel* agora precisa fazer é liberar a entrada da tabela de processos referente ao filho.

4.8.4 Implementação de EXEC

O código para EXEC segue o esboço da Figura 4-38. Ele está contido no procedimento *do_exec* (linha 17140). Depois de fazer algumas simples verificações de validade, o gerenciador de memória busca o nome do arquivo a ser executado a partir do espaço do usuário. Na linha 17172, envia uma mensagem especial para o sistema de arquivos, para alternar para o diretório do usuário, de modo que o caminho recém-buscado seja interpretado em relação ao diretório de trabalho do usuário em vez de em relação ao diretório de trabalho do gerenciador de memória.

Se o arquivo estiver presente e for executável, o gerenciador de memória lê o cabeçalho para extrair os tamanhos de segmento. Então, ele busca a pilha a partir do espaço do usuário (linhas 17188 e 17189), verifica se o novo processo pode compartilhar texto com um processo que já está executando (linha 17196), aloca memória para a nova imagem (linha 17199), corrige os ponteiros [veja as diferenças entre (b) e (c) na Figura 4-39] e lê os segmentos de texto (se necessário) e de dados (linhas 17221 a 17226). Por fim, ele processa os bits *setuid* e *setgid*, atualiza a entrada na tabela de processos e informa ao *kernel* que ele terminou, de modo que o processo possa ser agendado novamente

Embora o controle de todos os passos esteja em *do_exec*, muitos dos detalhes são executados por procedimentos subsidiários dentro de *exec.c*. *Read_header* (linha 17272), por exemplo, não apenas lê o cabeçalho e retorna os tamanhos de segmento, mas também verifica se o arquivo é um executável MINIX válido para o mesmo tipo de CPU para o qual o sistema operacional está compilado. Isso é feito por compilação condicional do teste apropriado no momento em que o gerenciador de memória é compilado (linhas 17322 a 17327). *Read_header* também verifica se todos segmentos ajustam-se no espaço de endereço virtual.

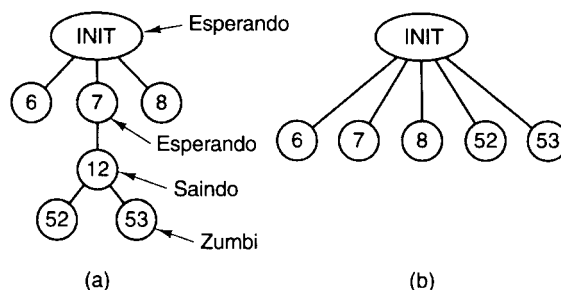



Figura 4-43 (a) A situação quando o processo 12 está para sair. (b) A situação depois que ele saiu.

O procedimento *new_mem* (linha 17366) verifica se há memória suficiente disponível para a nova imagem da memória. Ele procura por uma lacuna suficientemente grande para apenas os dados e a pilha, se o texto está sendo compartilhado; caso contrário, pesquisa uma única lacuna suficientemente grande para texto, para dados e para pilha combinados. Uma possível melhoria aqui seria pesquisar duas lacunas separadas, uma para o texto e outra para os dados e para a pilha, uma vez que não há nenhuma necessidade de essas áreas serem contíguas. Em versões anteriores do MINIX isso era exigido. Se memória suficiente for encontrada, a memória antiga é liberada e a nova memória adquirida. Se memória insuficiente estiver disponível, a chamada EXEC falha. Depois que a nova é alocada, *new_mem* atualiza o mapa de memória (em *mp_seg*) e informa isso ao *kernel* chamando o procedimento de biblioteca *sys_newmap*.

O restante de *new_mem* preocupa-se com zerar o segmento *bss*, a lacuna e o segmento de pilha. (O segmento *bss* é aquela parte do segmento de dados que contém todas as variáveis globais não-inicializadas). Muitos compiladores geram código explícito para zerar o segmento *bss*, mas fazer isso aqui permite que o MINIX funcione mesmo com compiladores que não o fazem. A lacuna entre os segmentos de dados e de pilha também é zerada, de modo que quando o segmento de dados é estendido por BRK, a memória adquirida conterá zeros. Isso é uma conveniência para o programador, que pode contar com novas variáveis que têm um valor inicial de zero, como também um recurso de segurança em um sistema operacional multiusuário, onde um processo previamente utilizando essa memória pode ter utilizado dados que não deveriam ser vistos por outros processos.

O próximo procedimento é *patch_ptr* (linha 17465), que faz o trabalho de realocar os ponteiros da Figura 4-39(b) para a forma da Figura 4-39(c). O trabalho é simples: ele examina a pilha para localizar todos os ponteiros e adiciona o endereço de base a cada um.

O procedimento *load_seg* (linha 17498) é chamado uma ou duas vezes por EXEC, para possivelmente carregar o segmento de texto e sempre carregar o segmento de dados. Em vez de simplesmente ler o arquivo bloco por bloco e, então, copiar os blocos para o usuário, um truque é utilizado para permitir que o sistema de arquivos carregue o segmento inteiro diretamente para o espaço do usuário. Com efeito, a chamada é decodificada pelo sistema de arquivos de uma maneira ligeiramente especial para que pareça ser uma leitura do segmento inteiro pelo próprio processo do usuário. Somente algumas poucas linhas no começo da rotina de leitura do sistema de arquivos sabem que um truque está em andamento aqui. O carregamento é significativamente acelerado com essa manobra.

O procedimento final em *exec.c* é *find_share* (linha 17535). Ele procura por um processo que pode compartilhar texto comparando o nó-i, o dispositivo e os tempos de modificação do arquivo a ser executado com aqueles dos processos existentes. Essa é uma pesquisa simples e direta dos campos apropriados em *mproc*. Naturalmente, ela deve ignorar o processo em nome do qual a pesquisa está sendo feita.

4.8.5 Implementação de BRK

Como acabamos de ver, o modelo de memória utilizado pelo MINIX é bem simples: a cada processo é dado uma única alocação contígua para seus dados e para sua pilha quando ele é criado. Ele nunca é movido na memória, nunca é comutado para fora da memória, nunca cresce e nunca encolhe. Tudo o que pode acontecer é que o segmento de dados pode consumir a lacuna a partir da extremidade inferior, e a pilha consumir a partir da extremidade superior. Sob essas circunstâncias, a implementação da chamada BRK em *break.c* é especialmente fácil. Consiste em verificar se os novos tamanhos são praticáveis e, então, atualizar as tabelas para refleti-los.

O procedimento de primeiro nível é *do_brk* (linha 17628), mas a maior parte do trabalho é feita em *adjust* (linha 17661). Este último verifica se os segmentos de pilha e de dados colidiram. Se isso tiver acontecido, a chamada BRK não poderá ser executada, mas o processo não será eliminado imediatamente. Um fator de segurança, *SAFETY_BYTES*, é adicionado ao topo do segmento de dados antes de fazer o teste, então (espera-se) a decisão de que a pilha cresceu demais pode ser tomada enquanto ainda há espaço suficiente na pilha para o processo continuar por um breve instante. Ele obtém o controle de volta (com uma mensagem de erro), e, então, pode imprimir mensagens apropriadas e desligar elegantemente.

Note que *SAFETY_BYTES* é definido, utilizando uma declaração **#define** no meio do procedimento (linha 17693). Essa utilização é bastante incomum: normalmente tais definições aparecem no começo de arquivos ou em arquivos de cabeçalho separados. O comentário associado revela que o programador achou que decidir sobre o tamanho do fator de segurança seria difícil. Não há dúvida de que essa definição foi feita assim para atrair atenção e, talvez, para estimular experimentação adicional.

A base do segmento de dados é constante, então, se *adjust* tiver de ajustar o segmento de dados, tudo o que ele tem a fazer é atualizar o campo de comprimento. A pilha cresce para baixo a partir de um ponto final fixo, assim, se *adjust* também notar que o ponteiro de pilha, que é dado para *adjust* como para um parâmetro, cresceu além do segmento de pilha (para um endereço mais baixo), tanto a origem como o comprimento são atualizados.

O último procedimento nesse arquivo, *size_ok* (linha 17736) faz um teste para ver se os tamanhos de segmento ajustam-se dentro do espaço de endereço, em cliques assim como em bytes. O código condicional para máquinas de 16 bits foi mantido na listagem para mostrar por que isso é escrito como uma função separada. Haveria pouco problema em ter isso como uma função separada para o MINIX de 32 bits. Ele é chamado somente em dois lugares, e colocar a linha 17765 no lugar das chamadas resultaria

em um código mais compacto, uma vez que as chamadas passam vários argumentos que não são utilizados na implementação de 32 bits.

4.8.6 Implementação da Manipulação de Sinais

Há oito chamadas de sistema que se relacionam com sinais, resumidas na Figura 4-44. Tais chamadas de sistema, assim como os próprios sinais, são processadas no arquivo *signal.c*. Uma chamada de sistema adicional, REBOOT, também é tratada por esse arquivo, uma vez que utiliza sinais para terminar todos os processos.

A chamada SIGACTION suporta as funções *sigaction* e *signal*, que permitem que um processo altere a maneira como ele responderá aos sinais. *Sigaction* é exigida pelo POSIX e é a chamada preferida para a maioria dos propósitos, mas a função de biblioteca *signal* é exigida pelo C padrão, e programas que devem ser portáveis para sistemas não-POSIX devem ser escritos, utilizando essa chamada. O código para *do_sigaction* (linha 17845) começa verificando um número de sinal válido e verificando que a chamada não é uma tentativa de alterar a resposta a um sinal SIGKILL (linhas 17851 e 17852). (Não é permitido ignorar, capturar ou bloquear SIGKILL. SIGKILL é o modo como, em última instância, o usuário pode controlar seus processos e um gerenciador de sistema pode controlar seus usuários.) SIGACTION é chamada com ponteiros para uma estrutura *sigaction*, *sig_osa*, que recebe os atributos de sinal antigos que estavam em efeito antes da chamada, e outra estrutura desse tipo, *sig_nsa*, que contém um novo conjunto de atributos.

O primeiro passo é chamar a tarefa de sistema para copiar os atributos atuais na estrutura apontada por *sig_osa*. SIGACTION poder ser chamada com um ponteiro *NULL* em *sig_nsa* para examinar os atributos antigos de sinal sem alterá-los. Nesse caso, *do_sigaction* retorna imediatamente (linha 17860). Se *sig_nsa* não for *NULL*, a estrutura que define a nova ação dos sinais é copiada para o espaço do gerenciador de memória. O código nas linhas 17867 a 17877 modifica os mapas de bits *mp_catch*, *mp_ignore* e *mp_sigpending* de acordo com o fato de a nova ação ser ignorar o sinal, utilizar o manipulador padrão ou capturar o sinal. As funções de biblioteca *sigaddset* e *sigdelset* são utilizadas apesar de as ações serem operações simples e diretas de manipulação de bits que poderiam ter sido implementadas com macros simples. Entretanto, tais funções são exigidas pelo padrão POSIX para tornar os programas que as utilizam facilmente portáveis, mesmo para sistemas em que o número de sinais excede o número de bits disponíveis em um inteiro. A utilização das funções de biblioteca ajuda a tornar o próprio MINIX facilmente portável para arquiteturas diferentes.

Por fim, são preenchidos os outros campos relacionados com sinais na parte do gerenciador de memória da tabela de processos. Para cada sinal possível, há um mapa de bits, *sa_mask*, que define quais sinais serão bloqueados enquanto um manipulador para esse sinal estiver executando. Para cada sinal, também há um ponteiro *sa_handler*. Esse pode conter um ponteiro para a função do manipulador ou valores especiais para indicar que o sinal deve ser ignorado ou ser tratado da maneira padrão. O endereço da rotina de biblioteca que invoca SIGRETURN quando o manipulador termina é armazenado em *mp_sigreturn*. Esse endereço é um dos campos na mensagem recebidos pelo gerenciador de memória.

O POSIX permite que um processo manipule o próprio tratamento de sinal mesmo enquanto dentro de um manipulador de sinal. Isso pode ser utilizado para alterar a resposta de sinal a sinais subseqüentes enquanto um sinal está sendo processado e, então, restaurar o conjunto de respostas normais. O próximo grupo de chamadas de sistema suporta esses recursos de manipulação de sinal. SIGPENDING é tratada por *do_sigpending* (linha 17889), que retorna o mapa de bits *mp_sigpending* para que um processo possa determinar se ele tem sinais pendentes. SIGPROCMASK, tratada por *do_sigprocmask*, retorna o conjunto de sinais que atualmente estão bloqueados e também pode ser utilizada para alterar o estado de um único sinal no conjunto ou para substituir o conjunto inteiro por um novo. O momento em que um sinal é desbloqueado é apropriado para verificar sinais pendentes e isso é feito por chamadas

Chamada de sistema	Propósito
SIGACTION	Modifica resposta para sinal futuro
SIGPROCMASK	Altera conjunto de sinais bloqueados
KILL	Envia sinal para outro processo
ALARM	Envia sinal de ALRM para si após intervalo
PAUSE	Suspende a si própria até sinal futuro
SIGSUSPEND	Altera conjunto de sinais bloqueados até então, PAUSE
SIGPENDING	Examina o conjunto de sinais pendentes (bloqueados)
SIGRETURN	Limpeza após manipulador de sinal

Figura 4-44 Chamadas de sistema relacionadas com sinais.

a *check_pending* na linha 17927 e na linha 17933. *Do_sigsuspend* (linha 17949) executa a chamada de sistema SIGSUSPEND. Essa chamada suspende um processo até que um sinal seja recebido. Como as outras funções que discutimos aqui, essa manipula mapas de bits. Ela também liga o bit SIGSUSPEND em *mp_flags*, que é o necessário para impedir a execução do processo. Novamente, este é um bom momento para fazer uma chamada a *check_pending*. Por fim, *do_sigreturn* trata SIGRETURN, que é utilizada para retornar de um manipulador personalizado. Ela restaura o contexto de sinal que existia quando o manipulador foi iniciado e também chama *check_pending* na linha 17980.

Alguns sinais como SIGINT, originam-se no próprio *kernel*. Esses sinais são tratados de uma maneira que é semelhante aos sinais gerados por um processo de usuário chamando KILL. Os dois procedimentos, *do_kill* (linha 17983) e *do_ksig* (linha 17994), são conceitualmente semelhantes. Os dois fazem com que o gerenciador de memória envie um sinal. Uma única chamada a KILL pode exigir entrega de sinais para um grupo de processos, e *do_kill* simplesmente chama *check_sig*, que verifica destinatários elegíveis na tabela de processos inteira. *Do_ksig* é chamada quando chega uma mensagem do *kernel*. A mensagem contém um mapa de bits que permite que o *kernel* gere múltiplos sinais com uma mensagem. Como com KILL, cada uma dessas pode precisar ser entregue para um grupo de processos. O mapa de bits é processado um bit por vez pelo laço nas linhas 18026 a 18048. Alguns sinais do *kernel* requerem atenção especial: o ID de processo é alterado em alguns casos para fazer com que o sinal seja entregue para um grupo de processos (linhas 18030 a 18033) e um SIGALRM é ignorado se não for solicitado. Com essa exceção, cada conjunto de bits resulta em uma chamada a *check_sig*, assim como em *do_kill*.

A chamada de sistema ALARM é controlada por *do_alarm* (linha 18056). Ela chama a próxima função, *set_alarm*, que envia uma mensagem à tarefa de relógio, dizendo-lhe para iniciar o temporizador. *Set_alarm* (linha 18067) é uma função separada, porque também é utilizada para desligar o temporizador quando um processo é encerrado com o temporizador ainda ligado. Quando o temporizador expira, o *kernel* anuncia o fato, enviando ao gerenciador de memória uma mensagem do tipo KSIG, que faz com que *do_ksig* execute como discutido acima. A ação padrão do sinal SIGALRM é eliminar o processo se o sinal não for capturado. Se SIGALRM é para ser capturado, um manipulador deve ser instalado por SIGACTION. A seqüência completa dos eventos para um sinal SIGALRM com um manipulador personalizado é mostrada na Figura 4-45. Há três seqüências de mensagens aqui. Nas mensagens (1), (2) e (3), o usuário faz uma chamada a ALARM via uma mensagem para o gerenciador de memória; o gerenciador envia uma solicitação para o relógio e este reconhece-a. Nas mensagens (4), (5) e (6), a tarefa de relógio envia o alarme para o gerenciador de memória, que chama a tarefa de sistema para preparar a pilha do processo de usuário para execução do manipulador de sinal [como na Figura

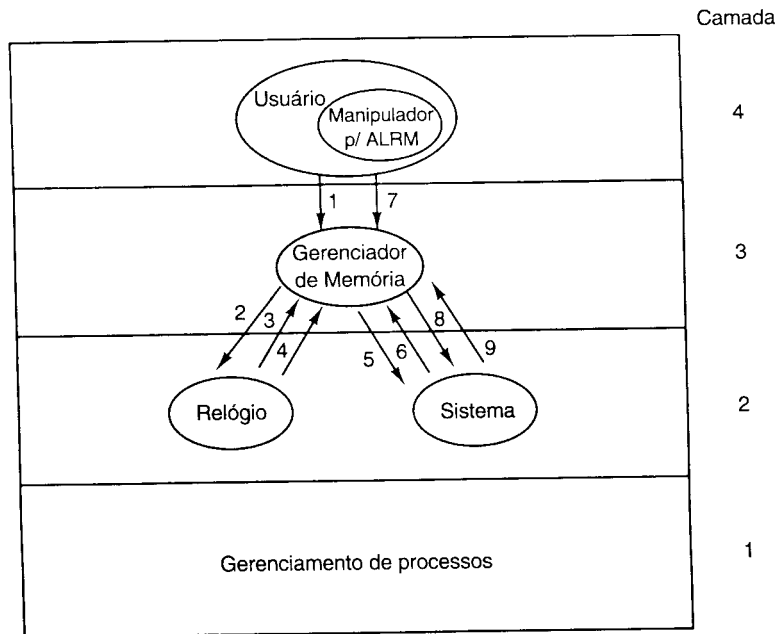



Figura 4-45 As mensagens para um alarme. As mais importantes são: (1) Usuário faz ALARM. (4) depois que o tempo configurado passou, o sinal chega. (7) O manipulador termina com uma chamada a SIGRETURN. Veja o texto para detalhes.

4-42(b)], e a tarefa de sistema responde. A mensagem (7) é a chamada a SIGRETURN que ocorre quando o manipulador completa a execução. Em resposta, o gerenciador de memória envia uma mensagem (8) à tarefa de sistema para fazê-la completar a limpeza, e a tarefa de sistema responde com uma mensagem (9). A mensagem (6) em si não causa a execução do manipulador, mas a seqüência será mantida porque a tarefa de sistema, como uma tarefa, terá permissão para completar seu trabalho devido ao algoritmo de agendamento de prioridade utilizado no MINIX. O manipulador é parte do processo do usuário e executará somente depois que a tarefa de sistema tiver concluído seu trabalho.

Do_pause cuida da chamada de sistema PAUSE (linha 18115). Tudo que é necessário é ligar um bit para impedir a resposta, mantendo assim o chamador bloqueado. O *kernel* nem mesmo precisa ser informado, uma vez que ele sabe que o chamador está bloqueado.

A última chamada de sistema tratada em *signal.c* é REBOOT (linha 18128). Essa chamada é utilizada somente por programas especializados executáveis pelo superusuário, mas serve para uma função importante. Ela assegura que todos os processos sejam encerrados de uma maneira ordenada e que o sistema de arquivos esteja sincronizado antes de a tarefa de sistema no *kernel* ser chamada para desligar. O encerramento de processos é feito utilizando *check_sig* para enviar um SIGKILL a todos os processos exceto *init*. Essa é a razão por que REBOOT é incluída nesse arquivo.

Várias funções de suporte em *signal.c* foram mencionadas de passagem. Agora as veremos em maiores detalhes. De longe, a mais importante é *sig_proc* (linha 18168), que realmente envia um sinal. Primeiro alguns testes são realizados. As tentativas de enviar para processos eliminados (linhas 18190 a 18192) ou pendurados (linhas 18194 a 18196) são problemas sérios que causam uma pane de sistema. Um processo que atualmente está sendo depurado é interrompido quando sinalizado (linhas 18198 a 18202). Se o sinal deve ser ignorado, o trabalho de *sig_proc* está completo na linha 18204. Essa é a ação-padrão para alguns sinais, por exemplo, os sinais que são exigidos pelo POSIX, mas não são suportados pelo MINIX. Se o sinal estiver bloqueado, a única ação que necessita ser executada é ligar um bit no mapa de bits *mp_sigpending* desse processo. O teste-chave (linha 18213) é distinguir processos que foram habilitados para capturar sinais daqueles que não foram. A essa altura, todas as outras considerações especiais foram eliminadas, e um processo que não pode capturar o sinal será encerrado.

Os sinais que são elegíveis para serem capturados são processados nas linhas 18214 a 18249. Uma mensagem é construída para ser enviada ao *kernel*, algumas partes da qual são cópias das informações na parte do gerenciador de memória da tabela de processos. Se o processo a ser sinalizado foi previamente suspenso por SIGSUSPEND, a máscara de sinal que foi salva no momento da suspensão é incluída na mensagem; caso contrário, a máscara atual de sinal é incluída (linhas 18213 a 18217). Outros itens incluídos na mensagem são vários endereços no espaço do processo sinalizado: o manipulador do sinal, o endereço da rotina de biblioteca *sigreturn* a ser chamada na conclusão do manipulador e o ponteiro atual de pilha.

Em seguida, é alocado espaço na pilha do processo. A Figura 4-46 mostra a estrutura que é colocada na pilha. A parte *sigcontext* é colocada na pilha para conservá-la para uma restauração posterior, uma vez que a estrutura correspondente na tabela de processos é alterada ao preparar-se para a execução do manipulador de sinal. A parte *sigframe* oferece um endereço de retorno para o manipulador de sinal e dados necessários para SIGRETURN completar a restauração do estado do processo quando o manipulador termina. O endereço de retorno e o ponteiro da moldura não são realmente utilizados por nenhuma parte do MINIX. Estão aí para enganar um depurador se qualquer pessoa alguma vez tentar rastrear a execução de um manipulador de sinal.

A estrutura a ser colocada na pilha do processo sinalizado é claramente grande. O código nas linhas 18225 e 18226 reserva espaço para ela, depois que uma chamada a *adjust* testa para ver se há espaço suficiente na pilha do processo. Se não houver espaço de pilha suficiente, o processo será eliminado, saltando para o rótulo *doterminate* usando o raramente utilizado goto de C (linhas 18228 e 18229).

Há um problema potencial com a chamada a *adjust*. Lembre-se de nossa discussão sobre a implementação de BRK, que *adjust* retorna um erro se a pilha estiver com *SAFETY_BYTES*, entrando no segmento de dados. A margem extra de erro é proporcionada porque a validade da pilha somente pode ser verificada ocasionalmente por software. Essa margem de erro é provavelmente excessiva no caso atual, uma vez que se sabe exatamente quanto espaço é necessário na pilha para o sinal, e o espaço adicional é necessário somente para o manipulador de sinal, presumivelmente uma função relativamente simples. É possível que alguns processos sejam terminados desnecessariamente porque a chamada a *adjust* falha. Isso é certamente melhor que ter programas que falham misteriosamente em outras ocasiões, mas o ajuste fino desses testes pode ser possível.

Se houver espaço suficiente na pilha, dois outros sinalizadores são verificados. O sinalizador *SA_NODEFER* indica se o processo sinalizado está bloqueando mais sinais do mesmo tipo ao tratar um sinal. O sinalizador *SA_RESETHAND* informa se o manipulador de sinal deve ser reinicializado ao receber esse sinal. (Isso oferece emulação confiável da antiga chamada *signal*. Embora esse "recurso" freqüentemente seja considerado uma falha na chamada antiga, suporte a recursos antigos exige suportar suas falhas também). O *kernel*, então, é notificado, utilizando a rotina de biblioteca *sys_sendsig* (linha 18242). Por fim, o bit que indica um sinal pendente é limpo, e *unpause* é chamada para terminar qualquer chamada de

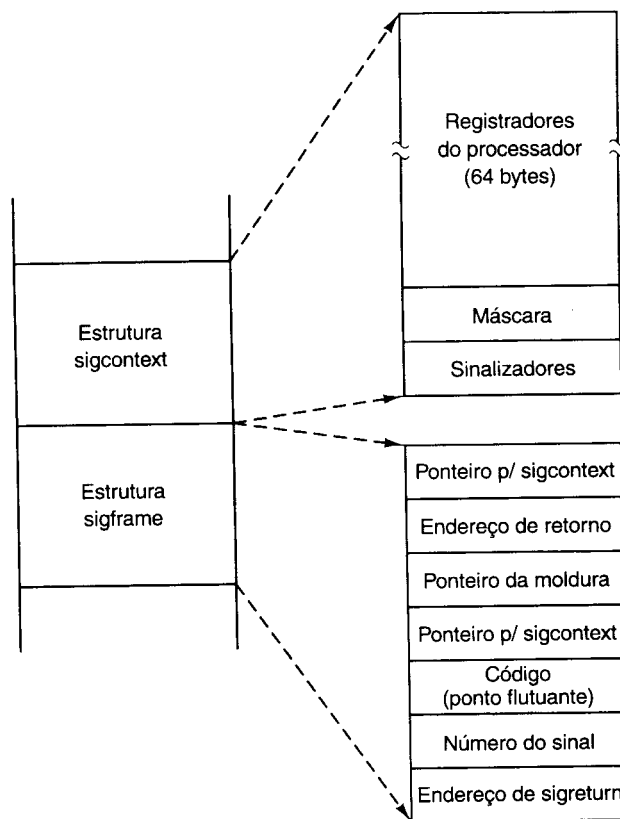



Figura 4-46 As estrutura *sigcontext* e *sigframe* colocadas na pilha para preparar para um manipulador de sinal. Os registradores do processador são uma cópia da moldura de pilha utilizada durante uma comutação de contexto.

sistema em que o processo possa estar pendurado. Quando o processo sinalizado executar em seguida, o manipulador de sinal executará.

Agora examinemos o código de encerramento, marcado pelo rótulo *doterminate* (linha 18250). O rótulo e um goto são a maneira mais fácil de tratar o possível fracasso da chamada *adjust*. Aqui, são processados os sinais que por uma razão ou outra não podem ou não devem ser capturados. A ação pode incluir um *dump* de núcleo, se isso for apropriado para o sinal e sempre acaba com o encerramento do processo como se ele tivesse saído, por meio de uma chamada a *mm_exit* (linha 18258).

Check_sig (linha 18265) é onde o gerenciador de memória verifica se um sinal pode ser enviado. A chamada

```
kill(0, sig)
```

faz com que o sinal indicado seja enviado para todos os processos no grupo do chamador (i. e., todos os processos iniciados do mesmo terminal). Os sinais originários do *kernel* e do REBOOT também podem afetar múltiplos processos. Por essa razão *check_sig*, faz um laço nas linhas 18288 a 18318, varrendo a tabela de processos para localizar todos os processos para os quais um sinal deve ser enviado. O laço contém um grande número de testes. Somente se todos eles tiverem sucesso, é que o sinal será enviado, chamando *sig_proc* na linha 18315.

Check_pending (linha 18330) é outra função chamada várias vezes no código que acabamos de revisar. Ela faz um laço por todos os bits no mapa de bits *mp_sigpending* para o processo referenciado por *do_sigmask*, *do_sigreturn* ou *do_sigsuspend*, para ver se qualquer sinal bloqueado tornou-se desbloqueado. Ela chama *sig_proc* para enviar o primeiro sinal desbloqueado que ela localizar. Uma vez que todos os manipuladores de sinal acabam causando a execução de *do_sigreturn*, isso basta para entregar por fim todos os sinais pendentes não-mascarados.

O procedimento *unpause* (linha 18359) tem a ver com sinais que são enviados a processos suspensos nas chamadas a PAUSE, WAIT, READ, WRITE ou SIGSUSPEND. PAUSE, WAIT e SIGSUSPEND podem ser verificadas consultando-se a parte do gerenciador de memória da tabela de processos, mas se nenhum desses for localizado, o sistema de arquivos deve

ser solicitado a utilizar sua própria função *do_unpause* para verificar uma possível queda em READ ou em WRITE. Em cada caso a ação é a mesma: uma resposta de erro é enviada para a chamada que espera, e o bit de sinalização que corresponde à causa da espera é zerado para que o processo possa reassumir sua execução e processar o sinal.

O procedimento final nesse arquivo é *dump_core* (linha 18402), que grava *dumps* de núcleo no disco. Um *dump* de núcleo consiste em um cabeçalho com as informações sobre o tamanho dos segmentos ocupados por um processo por uma cópia de todas as informações de status de um processo, obtidas copiando as informações referentes ao processo da tabela de processos do *kernel* e pela imagem da memória de cada um dos segmentos. Um depurador pode interpretar essas informações para ajudar o programador a determinar o que deu errado durante a execução do processo. O código para gravar o arquivo é simples e direto. O problema potencial mencionado na seção anterior novamente surge aqui, mas de uma forma um pouco diferente. Para assegurar que o segmento de pilha a ser registrado no *dump* de núcleo está atualizado, *adjust* é chamada na linha 18428. Essa chamada pode falhar por causa da margem de segurança embutida. O sucesso da chamada não é verificado por *dump_core*, então, o *dump* de núcleo será gravado em qualquer caso, mas dentro do arquivo as informações sobre a pilha podem estar incorretas.

4.8.7 Implementação das Outras Chamadas de Sistema

O arquivo *getset.c* contém um procedimento, *do_getset* (linha 18515), que executa as sete chamadas do gerenciador de memória restantes. Elas são mostradas na Figura 4-47. Todas são tão simples que não merecem um procedimento inteiro para cada uma. As chamadas GETUID e GETGID retornam o *uid* e o *gid* real ou efetivo.

A configuração de *uid* ou de *gid* é ligeiramente mais complexa do que a simples leitura. Uma verificação precisa ser feita para ver se o chamador é autorizado a configurar *uid* ou *gid*. Se o chamador passar no teste, o sistema de arquivos deve ser informado do novo *uid* ou *gid*, uma vez que a proteção de arquivos depende dele. A chamada SETSID cria uma nova sessão, e um processo que já é líder de grupo de processos não tem permissão para fazer isso. O teste na linha 18561 verifica isso. O sistema de arquivos completa o trabalho de tornar líder de sessão um processo sem controle do terminal.

Um suporte mínimo para depuração, por meio da chamada de sistema PTRACE, está no arquivo *trace.c*. Há 11 comandos que podem ser dados como um parâmetro para a chamada de sistema PTRACE. Eles são mostrados na Figura 4-48. No gerenciador de memória, *do_trace* processa quatro deles: *enable*, *exit*, *resume* e *step*. As solicitações para ativar ou para sair do rastreamento são completadas aqui. Todos os outros comandos são passados para a tarefa de sistema, que tem acesso à parte do *kernel* da tabela de processos. Isso é feito pela chamada à função de biblioteca *sys_trace* na linha 18669. Duas funções de suporte para rastreamento são oferecidas no fim de *trace.c*. *Stop_proc* é utilizado para interromper um processo rastreado quando ele é sinalizado, e *findproc* oferece suporte a *do_trace*, localizando na tabela de processos o processo a ser rastreado.

4.8.8 Utilitários do Gerenciador de Memória

Os arquivos restantes contêm tabelas e rotinas utilitárias. O arquivo *alloc.c* é onde o sistema monitora quais partes da memória estão em uso e quais estão livres. Ele tem quatro pontos de entrada:

1. *alloc_mem* – solicita um bloco de memória de um dado tamanho.
2. *free_mem* – retorna memória que não é mais necessária.
3. *max_hole* – calcula o tamanho da maior lacuna disponível.
4. *mem_init* – inicializa a lista de livres quando o gerenciador de memória inicia sua execução.

Como dissemos antes, *alloc_mem* (linha 18840) simplesmente utiliza primeiro ajuste em uma lista de lacunas classificadas por endereço de memória. Se encontrar um pedaço que é muito grande, ela toma o que precisa e deixa

Chamada de sistema	Descrição
GETUID	Retorna o *uid* real e efetivo
GETGID	Retorna o *gid* real e efetivo
GETPID	Retorna os *pids* do processo e seu pai
SETUID	Configura o *uid* real e efetivo do chamador
SETGID	Configura o *gid* real e efetivo do chamador
SETSID	Cria nova sessão, retorna o *pid*
GETPRP	Retorna o ID do grupo do processo

Figura 4-47 As chamadas de sistema suportadas em *mm/getset.c*.

Comando	Descrição
T_ STOP	Pára o processo
T_OK	Habilita o rastreamento desse processo pelo pai
T_GETINS	Retorna valor do espaço de texto (instruções)
T_GETDATA	Retorna valor do espaço de dados
T_GETUSER	Retorna valor da tabela de processos do usuário
T_SETINS	Configura um valor no espaço de instruções
T_SETDATA	Configura um valor no espaço de dados
T_SETUSER	Configura um valor na tabela de processos do usuário
T_RESUME	Reassume a execução
T_EXIT	Sai
T_STEP	Configura o bit de rastreio

Figura 4-48 Comandos de depuração suportados por *mm/trace.c.*

o restante na lista de livres, mas reduzido em tamanho pela quantidade tomada. Se uma lacuna inteira é necessária, *del_slot* (linha 18926) é chamada para remover a entrada da lista de livres.

O trabalho de *free_mem* é verificar se um pedaço recentemente liberado de memória pode ser fundido com lacunas de qualquer um dos lados. Se puder, *merge* (linha 18949) é chamado para unir as lacunas e atualizar as listas.

Max_hole (linha 18985) varre a lista de lacunas e retorna o maior item que encontrar. *Mem_init* (linha 19005) constrói a lista de livres inicial, consistindo em toda a memória disponível.

O próximo arquivo é o *utility.c*, que armazena alguns procedimentos diversos utilizados em vários lugares no gerenciador de memória. O procedimento *allowed* (linha 19120) verifica se um dado acesso é permitido para um arquivo. Por exemplo, *do_exec* precisa saber se um arquivo é executável.

O procedimento *no_sys* (linha 19161) nunca dever ser chamado. Ele é oferecido somente no caso de um usuário em algum momento chama o gerenciador de memória com um número de chamada de sistema que é inválido ou que não é tratado pelo gerenciador de memória.

Panic (linha 19172) é chamado somente quando o gerenciador de memória detectou um erro do qual não pode recuperar-se. Ele informa o erro à tarefa de sistema, que, então, causa uma suspensão brusca do MINIX. *Panic* não é chamado incolumemente.

A última função em *utility.c* é *tell_fs*, que constrói uma mensagem e envia-a para o sistema de arquivos, quando este último precisa ser informado dos eventos tratados pelo gerenciador de memória.

Os dois procedimentos no arquivo *putk.c* também são utilitários, embora de um caráter bem diferente dos anteriores. De vez em quando, chamadas a *printf* são inseridas no gerenciador de memória, principalmente para depuração. Além disso, *panic* chama *printf*. Como já mencionado, o nome *printf* é realmente uma macro definida como *printk*, de modo que chamadas a *printf* não utilizam o procedimento-padrão de biblioteca de E/S que envia mensagens para o sistema de arquivos. *Printk* chama *putk* para comunicar-se diretamente com a tarefa de terminal, algo que é proibido para usuários comuns. Vimos uma rotina de mesmo nome no código do *kernel*.

4.9 RESUMO

Neste capítulo, examinamos o gerenciamento de memória, tanto em geral como no MINIX. Vimos que os sistemas mais simples não fazem troca em disco (*swap*) ou paginação. Uma vez que um programa é carregado na memória, ele fica aí até terminar. Alguns sistemas operacionais permitem somente um processo por vez na memória, enquanto outros suportam multiprogramação.

O próximo passo é a troca em disco. Quando a troca em disco é utilizada, o sistema pode tratar mais processos além do espaço disponível na memória. Os processos para os quais não há espaço são trocados para o disco. Espaço livre na memória e em disco pode ser monitorado com um mapa de bits ou de uma lista de lacunas.

Computadores mais avançados, freqüentemente, têm alguma variante de memória virtual. Na forma mais simples, o espaço de endereços de cada processo é dividido em blocos de tamanho uniforme chamados páginas, que podem ser colocadas em qualquer moldura de página disponível na memória. Há muitos algoritmos de substituição de páginas, dois dos melhores conhecidos são o de segunda chance e o de idade. Para fazer os sistemas de paginação funcionar bem, não é suficiente escolher um algoritmo: é necessária atenção a questões como determinar o conjunto funcional, a política de alocação de memória e o tamanho de página.

A segmentação ajuda no tratamento de estruturas de dados que mudam de tamanho durante a execução e simplifica a vinculação e o compartilhamento. Também facilita oferecer proteção diferente para segmentos diferentes. Às vezes, segmentação e paginação são combinadas para oferecer uma memória virtual de duas dimensões. O sistema MULTICS e o Pentium da Intel suportam segmentação e paginação.

O gerenciamento de memória no MINIX é simples. A memória é alocada quando um processo executa uma chamada de sistema FORK ou EXEC. A memória assim alocada nunca é aumentada ou diminuída durante a vida do processo. Nos processadores Intel, há dois modelos de memória utilizados pelo MINIX. Os programas pequenos podem ter instruções e dados no mesmo segmento da memória. Programas maiores utilizam espaços de instruções e de dados separados (I e D separados). Os processos com espaços I e D separados podem compartilhar a parte de texto de sua memória, portanto, durante um FORK deve ser alocada memória apenas para dados e para pilha. Isso também pode ser verdadeiro durante um EXEC se outro processo já estiver utilizando o texto necessário pelo novo programa.

A maior parte do trabalho do gerenciador de memória não é dedicada a monitorar a memória livre, o que ele faz utilizando uma lista de lacunas e o algoritmo do primeiro ajuste, mas sim, a executar as chamadas de sistema relacionadas com o gerenciamento de memória. Diversas chamadas de sistema suportam sinais no estilo POSIX e, uma vez que a ação-padrão da maioria dos sinais é encerrar o processo sinalizado, é apropriado tratá-los no gerenciador de memória, que inicia o encerramento de todos os processos. Várias chamadas de sistema não diretamente relacionadas com memória também são tratadas pelo gerenciador de memória, principalmente porque ele é menor que o sistema de arquivos e assim foi mais conveniente colocá-la aqui.

EXERCÍCIOS

1. Um sistema de computador tem espaço suficiente para armazenar quatro programas em sua memória principal. Esses programas ficam inativos esperando E/S metade do tempo. Que fração de tempo da CPU é desperdiçada?

2. Considere um sistema de troca em disco no qual a memória consiste nos seguintes tamanhos de lacuna pela ordem de memória: 10K, 4K, 20K, 18K, 7K, 9K, 12K e 15K. Que lacuna é tomada para sucessivas solicitações de segmento de

(a) 12K
(b) 10K
(c) 9K

no caso do algoritmo de primeiro ajuste? Agora repita a pergunta para melhor ajuste, pior ajuste e próximo ajuste.

3. Qual é a diferença entre um endereço físico e um endereço virtual?

4. Utilizando a tabela de páginas da Figura 4-8, forneça o endereço físico correspondente a cada um dos seguintes endereços virtuais:

(a) 20
(b) 4100
(c) 8300

5. O processador Intel 8086 não suporta memória virtual. Contudo, algumas empresas chegaram a comercializar sistemas que continham uma CPU 8086 não-modificada e que faziam paginação. Faça uma suposição elaborada de como eles faziam isso. (Sugestão: pense na posição lógica da MMU.)

6. Se uma instrução leva 1 microssegundo, e uma falha de página leva mais n microssegundos, forneça uma fórmula para o tempo efetivo de instrução se falhas de página ocorrerem a cada k instruções.

7. Uma máquina tem um espaço de endereço de 32 bits e páginas de 8K. A tabela de páginas está inteiramente em hardware, com uma palavra de 32 bits por entrada. Quando um processo inicia, a tabela de páginas é copiada para o hardware a partir da memória a uma taxa de uma palavra a cada 100ns. Se cada processo executa por 100ms (incluindo o tempo de carregar a tabela de página), que fração de tempo da CPU é dedicada para carregar as tabelas de páginas?

8. Um computador com endereços de 32 bits utiliza uma tabela de página de dois níveis. Endereços virtuais são divididos em um campo de 9 bits da tabela de páginas de primeiro nível, um campo de 11 bits da tabela de páginas de segundo nível e um deslocamento. Qual é o tamanho das páginas e quantas existem no espaço de endereços?

9. A seguir é apresentada a listagem de um programa curto de linguagem *assembly* para um computador com páginas de 512 bytes. O programa está localizado no endereço 1020, e seu ponteiro de pilha está em 8192 (a pilha cresce em direção a 0). Forneça a cadeia de referências de páginas gerada por esse programa. Cada instrução ocupa 4 bytes (1 palavra) e referências tanto de instruções como de dados contam na cadeia de referências.

Carregue a palavra 6144 no registrador 0
Coloque o registrador 0 na pilha
Chame um procedimento em 5120, colocando o endereço de retorno na pilha
Subtraia a constante imediata 16 do ponteiro da pilha
Compare o parâmetro real com a constante imediata 4
Salte se igual a 5152

10. Suponha que um endereço virtual de 32 bits seja quebrado em quatro campos, a, b, c e d. Os primeiros três são utilizados para um sistema de tabela de páginas de três níveis. O quarto campo, d, é o deslocamento. O número de páginas depende dos tamanhos de todos os quatro campos? Se não, quais importam e quais não?

11. Um computador cujos processos têm 1.024 páginas em seu espaço de endereços mantém suas tabelas de páginas na memória. O *overhead* exigido para ler uma palavra da tabela de páginas é 500ns. Para reduzir esse *overhead*, o computador tem um TLB, que armazena 32 pares (página virtual, moldura de página física) e pode fazer uma busca em 100ns. Qual a taxa de acertos necessária para reduzir o *overhead* médio para 200ns?

12. O TLB no VAX não contém um bit *R*. Por quê?

13. Uma máquina tem endereços virtuais de 48 bits e endereços físicos de 32 bits. As páginas são de 8K. Quantas entradas são necessárias na tabela de páginas?

14. Um computador tem quatro molduras de página. O tempo de carregamento o tempo do último acesso e os bits *R* e *M* para cada página são como mostrado a seguir (os tempos estão em tiques do relógio):

Página	Carregado	Última ref.	R	M
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

(a) Que página NRU substituirá?
(b) Que página FIFO substituirá?
(c) Que página LRU substituirá?
(d) Que página o algoritmo de segunda chance substituirá?

15. Se a substituição de página FIFO for utilizada com quatro molduras de página e oito páginas, quantas falhas de página ocorrerão com a cadeia de referências 0172327103 se as quatro molduras estão inicialmente vazias? Agora repita esse problema para LRU.

16. Um computador pequeno tem quatro molduras de página. No primeiro tique de relógio, os bits *R* são 0111 (a página 0 é 0, as restantes são 1). Em subseqüentes tiques de relógio, os valores são 1011, 1010, 1101, 0010, 1010, 1100 e 0001. Se o algoritmo de idade for utilizado com um contador de 8 bits, forneça os valores dos quatro contadores depois do último tique.

17. Quanto tempo leva para carregar um programa de 64K de um disco cujo tempo médio de busca é 30ms, seu tempo de rotação é 20ms e cujas trilhas armazenam 32K?
(a) para um tamanho de página 2K?
(b) para um tamanho de página 4K?

As páginas são distribuídas aleatoriamente no disco.

18. Uma das primeiras máquinas de compartilhamento de tempo, o PDP-1, tinha uma memória de 4K palavras de 18 bits. Ele mantinha um processo por vez na memória. Quando o agendador decidia executar outro processo, o processo na memória era gravado em um tambor de paginação, com 4K palavras de 18 bits em torno da circunferência do tambor. O tambor poderia iniciar gravação (ou leitura) em qualquer palavra, em vez de somente na palavra 0. Por que você supõe que esse tambor foi escolhido?

19. Um computador oferece a cada processo 65.536 bytes de espaço de endereço dividido em páginas de 4096 bytes. Um programa particular tem um tamanho de texto de 32.768 bytes, um tamanho de dados de 16.386 bytes e um tamanho de pilha de 15.870 bytes. Esse programa ajustar-se-á no espaço de endereço? Se o tamanho de página fosse 512 bytes, ele se ajustaria? Lembre-se de que uma página não pode conter partes de dois segmentos diferentes.

20. Foi observado que o número de instruções executadas entre falhas de página é diretamente proporcional ao número de molduras de página alocadas para um programa. Se a memória disponível é dobrada, o intervalo médio entre falhas de página também dobra. Suponha que uma instrução normal leve 1 microssegundo, mas se uma falha de página ocorre, ela leva 2.001 microssegundos (i. e., 2ms para tratar a falha). Se um programa levasse 60 s para executar, tempo durante o qual ocorressem 15.000 falhas de página, quanto levaria para executar se o dobro da memória estivesse disponível?

21. Um grupo de projetistas de sistema operacional da Companhia Frugal de Computadores está pensando sobre maneiras de reduzir a quantidade de espaço necessário para seu novo sistema operacional. O guru-chefe sugeriu apenas não se incomodar em salvar o texto de programa ao fazer uma troca em disco, mas simplesmente paginá-lo diretamente a partir do arquivo binário sempre que for necessário. Há algum problema com essa abordagem?

22. Explique a diferença entre fragmentação interna e fragmentação externa. Qual ocorre em sistemas de paginação? Qual ocorre em sistemas que utilizam segmentação pura?

23. Quando tanto segmentação como paginação são utilizadas, como no MULTICS, primeiro o descritor de segmento deve ser pesquisado, depois o descritor de página. O TLB também funciona dessa maneira, com dois níveis de pesquisa?

24. Por que o esquema de gerenciamento de memória do MINIX torna necessário ter um programa como *chmem*?

25. Modifique o MINIX para liberar a memória de um zumbi assim que ele entra no estado zumbi, em vez de aguardar até que o pai comece a esperá-lo.

26. Na implementação atual do MINIX, quando uma chamada de sistema EXEC é feita, o gerenciador de memória verifica se uma lacuna suficientemente grande para conter a nova imagem da memória está atualmente disponível. Se não estiver, a chamada é rejeitada. Um algoritmo melhor seria ver se uma lacuna suficientemente grande estaria disponível depois que a imagem atual de memória fosse liberada. Implemente esse algoritmo.

27. Quando executa uma chamada de sistema EXEC, o MINIX utiliza um truque para fazer o sistema de arquivos ler segmentos inteiros de uma vez. Invente e implemente um truque semelhante para permitir que *dumps* de núcleo sejam gravados de uma maneira semelhante.

28. Modifique o MINIX para fazer troca em disco.

29. Na Seção 4.7.5, foi indicado que, em uma chamada EXEC, testando-se para uma lacuna adequada antes de liberar a memória do processo atual, uma implementação subótima é alcançada. Reprograme esse algoritmo para fazer melhor.

30. Na Seção 4.8.4, foi indicado que seria melhor pesquisar lacunas para os segmentos de texto e de dados separadamente. Implemente esse aprimoramento.

31. Redesenhe *adjust* para evitar o problema de os processos sinalizados serem eliminados desnecessariamente por causa de um teste muito estrito para espaço de pilha.

5

Sistemas de Arquivos

Todo aplicativo de computador precisa armazenar e recuperar informações. Enquanto um processo está executando, ele pode armazenar uma quantidade limitada de informações dentro de seu próprio espaço de endereços. Entretanto, a capacidade de armazenamento é limitada ao tamanho do espaço de endereço virtual. Para alguns aplicativos, esse tamanho é adequado, mas para outros, como reservas de linha aérea, sistemas bancários ou registro geral de grandes corporações, é, de longe, muito pequeno.

Um segundo problema com manter as informações dentro do espaço de endereços de um processo, é que quando o processo termina, as informações são perdidas. Para muitos aplicativos (p. ex., para bancos de dados), as informações devem ser mantidas durante semanas, meses ou mesmo eternamente. É inaceitável que elas desapareçam quando o processo que as utiliza termina. Além disso, elas não devem desaparecer quando uma falha do computador "mata" o processo.

Um terceiro problema: freqüentemente é necessário que múltiplos processos acessem as informações ou parte das informações ao mesmo tempo. Se temos uma lista de telefones *on-line* armazenada no espaço de endereços de um único processo, somente aquele processo pode acessá-la. A maneira de resolver esse problema é tornar as próprias informações independentes de qualquer processo.

Assim, temos três exigências essenciais para armazenamento de informações de longo prazo:

1. Deve ser possível armazenar uma grande quantidade de informações.
2. A informação deve sobreviver à finalização do processo que a utiliza.
3. Múltiplos processos devem ser capazes de acessar as informações concorrentemente.

A solução normal para todos esses problemas é armazenar as informações em discos e em outras mídias externas, em unidades chamadas **arquivos**. Os processos, então, podem lê-los e gravar novas informações se necessário. As informações armazenadas em arquivos devem ser **persistentes**, isto é, não devem ser afetadas pela criação e pela finalização de processos. Um arquivo deve desaparecer apenas quando seu proprietário explicitamente removê-lo.

Os arquivos são gerenciados pelo sistema operacional. O modo como eles são estruturados, nomeados, acessados, utilizados, protegidos e implementados constitui temas importantes no projeto de um sistema operacional. Como um todo, a parte do sistema operacional que lida com arquivos é conhecida como o **sistema de arquivos**, sendo o assunto deste capítulo.

Do ponto de vista do usuário, o aspecto mais importante de um sistema de arquivos é como aparece para ele, isto é, o que constitui um arquivo, como os arquivos são nomeados e protegidos, que operações são permitidas em arquivos e assim por diante. Os detalhes de como listas encadeadas ou mapas de bits são utilizados para registrar o espaço livre e quantos setores estão em um bloco lógico são de interesse menor, embora sejam de grande importância para os projetistas do sistema de arquivos. Por essa razão, estruturamos o capítulo em várias seções. As duas primeiras são dedicadas à interface do usuário para arquivos e para diretórios, respectivamente. Então, vem uma detalhada discussão sobre como é implementado o sistema de arquivos. Depois, veremos mecanismos de proteção e de segurança em sistemas de arquivos. Por fim, será enfocado o sistema de arquivos.

5.1 ARQUIVOS

Nesta seção, veremos arquivos do ponto de vista do usuário, isto é, como eles são utilizados e quais são suas propriedades.

5.1.1 Nomes de Arquivo

Os arquivos são um mecanismo de abstração. Oferecem uma maneira de armazenar as informações no disco e de lê-las de volta mais tarde. Isso deve ser feito de tal maneira que esconda do usuário os detalhes de como e onde as informações são armazenadas e de como os discos realmente trabalham.

Provavelmente, a mais importante característica de qualquer mecanismo de abstração é a maneira como são nomeados os objetos que estão sendo gerenciados. Assim, iniciaremos nosso exame dos sistemas de arquivos com o tema atribuição de nomes de arquivo. Quando um processo cria um arquivo, ele lhe dá um nome. Quando o processo termina, o arquivo continua a existir e a poder ser acessado por outros processos, utilizando seu nome.

As regras exatas para nomes de arquivos variam um pouco de um sistema para outro, mas todos os sistemas operacionais permitem cadeias de uma a oito letras como nomes de arquivo válidos. Assim *andrea*, *bruce* e *cathy* são nomes possíveis de arquivo. Freqüentemente, algarismos e caracteres especiais também são permitidos, então, nomes como *2*, *urgente!* e *Figura 2-14* freqüentemente são válidos também. Muitos sistemas de arquivos suportam nomes com até 255 caracteres.

Alguns sistemas de arquivos distinguem entre nomes escritos com letras maiúsculas e nomes escritos com letras minúsculas, enquanto outros não o fazem. O UNIX entra na primeira categoria; o MS-DOS entra na segunda. Assim um sistema UNIX pode ter todos os seguintes nomes como arquivos distintos: *barbara*, *Barbara*, *BARBARA*, *BARbara* e *BarBaRa*. No MS-DOS, todos eles designam o mesmo arquivo.

Muitos sistemas operacionais suportam nomes de arquivo de duas partes, ambas separadas por um ponto, como em *prog.c*. A parte que se segue ao ponto é chamada extensão de arquivo e normalmente indica algo sobre o arquivo. No MS-DOS, por exemplo, nomes de arquivo têm de 1 a 8 caracteres, com mais uma extensão opcional de 1 a 3 caracteres. No UNIX, o tamanho da extensão, se houver uma, é o usuário quem determina, e um arquivo pode ter até duas ou mais extensões, como em *prog.c.Z*, onde *Z* é comumente utilizado para indicar que o arquivo (o *prog.c*) foi compactado, utilizando o algoritmo de compactação de Ziv-Lempel. Algumas extensões de arquivo mais comuns e seus significados são mostrados na Figura 5-1.

Em alguns casos, as extensões de arquivo são apenas convenções e não são necessáriamente impostas. Um arquivo chamado *arquivo.txt* é provavelmente algum tipo de arquivo de texto, mas esse nome é mais para lembrar o proprietário do que para carregar quaisquer informações específicas para o computador. Por outro lado, um compilador C, ao compilar, pode realmente insistir em que os arquivos sejam terminados em *.c* e pode recusar-se a compilá-los se essa exigência não for atendida.

Convenções como essa são especialmente úteis quando o mesmo programa pode tratar vários tipos diferentes de arquivos. O compilador C, por exemplo, pode receber uma lista de vários arquivos a compilar e a vincular juntos, sendo alguns deles arquivos em C e, outros, arquivos em linguagem *assembly*. A extensão, então, torna-se essencial para informar ao compilador quais são arquivos em C, quais são em *assembly* e quais são outros arquivos.

5.1.2 Estruturas de Arquivos

Os arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns são representadas na Figura 5-2. O arquivo na Figura 5-2(a) é uma seqüência de bytes não-estruturada. Com efeito, o sistema operacional

Extensão	Significado
.bak	Arquivo de backup
.c	Programa fonte em C
.f77	Programa Fortran 77
.gif	Imagem no formato GIF (Graphical Interchange Format, da CompuServe)
.hlp	Arquivo de ajuda
.html	Documento HTML (HyperText Markup Language) da WWW (World Wide Web)
.mpg	Filme codificado com o padrão MPEG
.o	Arquivo-objeto (saída de compilador, ainda não linkeditado)
.ps	Arquivo PostScript
.tex	Entrada para o programa de formatação TEX
.txt	Arquivo genérico de texto
.zip	Arquivo compactado

Figura 5-1 Algumas típicas extensões de arquivo.

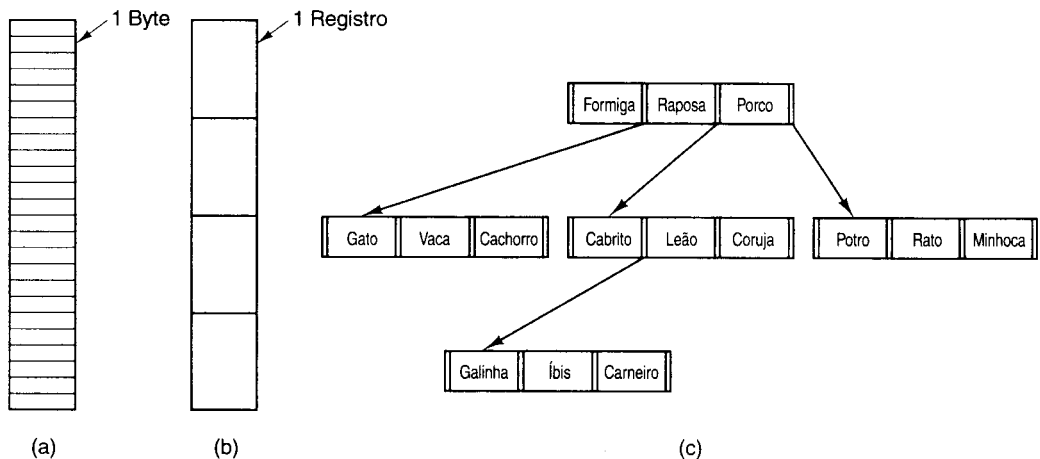



Figura 5-2 Três tipos de arquivos. (a) Seqüência de bytes. (b) Seqüência de registros. (c) Árvore.

não sabe nem se importa com o que está no arquivo. Tudo que ele vê são bytes. Qualquer significado deve ser imposto por programas no nível do usuário. Tanto o UNIX como o MS-DOS utilizam essa abordagem. A propósito, o WINDOWS 95 utiliza basicamente o sistema de arquivos do MS-DOS, com uma pequena adição sintática (p. ex., nomes de arquivo longos). Então, quase tudo dito neste capítulo sobre o MS-DOS também se aplica ao WINDOWS 95. O WINDOWS NT, porém, é completamente diferente.

Ter o sistema operacional considerando arquivos como nada mais do que seqüências de bytes oferece um máximo de flexibilidade. Os programas de usuário podem colocar qualquer coisa que quiserem em arquivos e nomeá-los de qualquer maneira que lhe seja conveniente. O sistema operacional não ajuda, mas também não atrapalha. Para usuários que querem fazer coisas incomuns, este último aspecto pode ser muito importante.

O primeiro passo na estrutura é mostrado na Figura 5-2(b). Neste modelo, um arquivo é uma seqüência de registros de comprimento fixo, cada um com alguma estrutura interna. O cerne da idéia de que um arquivo é uma seqüência de registros está no fato de que a operação de leitura retorna um registro, e as operações de gravação sobrescrevem ou anexam um registro. Antigamente, quando reinavam os cartões perfurados de 80 colunas, muitos sistemas operacionais baseavam seus sistemas de arquivos em arquivos que consistiam em registros de 80 caracteres, que, de fato, representavam imagens de cartões. Esses sistemas também suportavam arquivos de registros com 132 caracteres, que foram projetados para impressoras de linha (que nesse tempo eram grandes impressoras de cadeia com 132 colunas). Os programas liam a entrada em unidades de 80 caracteres e gravavam em unidades de 132 caracteres, embora os 52 finais pudessem ser espaços, naturalmente.

Um (antigo) sistema que via arquivos como seqüências de registros de comprimento fixo era o CP/M. Ele utilizava um registro de 128 caracteres. Hoje em dia, a idéia de um arquivo como uma seqüência de registros de comprimento fixo foi completamente abandonada, embora um dia tenha sido a norma.

O terceiro tipo de estrutura de arquivos é mostrado na Figura 5-2(c). Nessa organização, um arquivo consiste em uma árvore de registros, não necessariamente todos do mesmo comprimento, cada um contendo um **campo-chave** em uma posição fixa no registro. A árvore é classificada pelo campo-chave, permitindo localizar rapidamente uma chave particular.

A operação básica aqui não é obter o "próximo" registro, embora isso também seja possível, mas obter o registro com uma chave específica. Para o arquivo do zoológico da Figura 5-2(c), poderia ser solicitado que o sistema obtivesse o registro cuja chave é *potro*, por exemplo, sem se preocupar com sua posição exata no arquivo. Além disso, novos registros podem ser adicionados ao arquivo, com o sistema operacional e não com o usuário, decidindo onde colocá-los. Esse tipo de arquivo é claramente bem diferente dos fluxos de byte não-estruturados utilizados no UNIX e no MS-DOS, mas é amplamente utilizado nos *mainframes* de grande porte ainda usados em algum processamento comercial de dados.

5.1.3 Tipos de Arquivo

Muitos sistemas operacionais suportam vários tipos de arquivos. O UNIX e o MS-DOS, por exemplo, têm arquivos e diretórios comuns. O UNIX também tem arquivos de caractere e de bloco especiais. **Arquivos comuns** são os que contêm informações do usuário. Todos os arquivos da Figura 5-2 são comuns. **Diretórios** são arquivos de sistema para manter a estrutura do sistema de arquivos. Estudaremos diretórios a seguir. **Arquivos especiais de caractere** relacionam-se com a entrada/saída e são utilizados para mo-

delar dispositivos de E/S seriais como terminais, impressoras e redes. **Arquivos especiais de bloco** são utilizados para modelar discos. Neste capítulo, estaremos interessados principalmente em arquivos comuns.

Arquivos comuns são geralmente arquivos ASCII ou arquivos binários. Os arquivos ASCII consistem em linhas de texto. Em alguns sistemas, cada linha é terminada por um caractere de retorno de carro. Em outros, o caractere de quebra de linha é utilizado. Ocasionalmente, ambos são exigidos. As linhas não necessitam ser todas do mesmo comprimento.

A grande vantagem de arquivos ASCII é que podem ser exibidos e impressos como são e podem ser editados com um editor de texto comum. Além disso, se um grande número de programas utiliza arquivos ASCII para entrada e saída, é fácil conectar a saída de um programa à entrada de outro, como em canalizações (*pipelines*) do *shell*. (O "encanamento" interprocesso não é nada fácil, mas interpretar a informação certamente é, se uma convenção-padrão, como ASCII, for utilizada para expressá-la.)

Outro tipo são arquivos binários, o que significa simplesmente que eles não são arquivos ASCII. Imprimi-los resulta em uma lista incompreensível cheia de, aparentemente, lixo aleatório. Normalmente, eles têm alguma estrutura interna.

Por exemplo, na Figura 5-3(a) vemos um arquivo binário executável simples, obtido de uma versão anterior do UNIX. Embora tecnicamente o arquivo seja somente uma seqüência de bytes, o sistema operacional somente executará um arquivo se tiver o formato adequado. Ele tem cinco seções: cabeçalho, texto, dados, bits de realocação e tabela de símbolos. O cabeçalho inicia com o chamado **número mágico**, identificando o arquivo como um arquivo executável (para evitar a execução acidental de um arquivo que não tenha esse formato). Então, vêm inteiros de 16 bits fornecendo os tamanhos das várias partes do arquivo, o endereço em que a execução inicia e alguns bits de sinalização. Seguindo-se ao cabeçalho estão o texto e dados do próprio programa. Estes últimos são carregados na memória e realocados utilizando os bits de realocação. A tabela de símbolos é utilizada para depuração.

Nosso segundo exemplo de arquivo binário também é um arquivo do UNIX. Consiste em uma coleção de procedimentos de biblioteca (módulos) compilados, mas não-

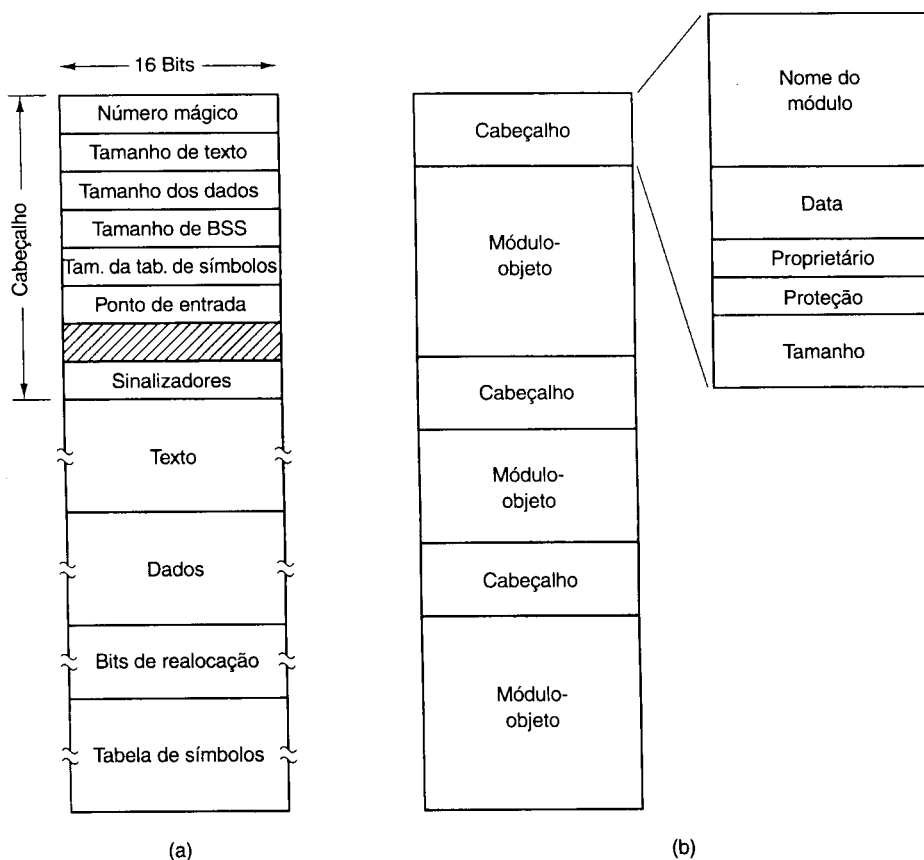



Figura 5-3 (a) Um arquivo executável. (b) Um arquivo.

linkeditados. Cada um deles é prefaciado por um cabeçalho que informa seu nome, sua data de criação, seu proprietário, seu código de proteção e seu tamanho. Assim como com o arquivo executável, os cabeçalhos de módulo estão cheios de números binários. Copiá-los para a impressora produziria simplesmente lixo.

Todo sistema operacional deve reconhecer um tipo de arquivo, seu próprio arquivo executável, mas alguns reconhecem mais. O antigo sistema TOPS-20 ia tão longe a ponto de examinar a data/hora de criação de qualquer arquivo a ser executado. Então, ele localizava o arquivo-fonte e via se o fonte tinha sido modificado desde que o binário foi criado. Se tivesse, automaticamente recompilava o fonte. Em termos do UNIX, o programa *make* foi construído dentro do *shell*. Como as extensões de arquivo eram obrigatórias, o sistema operacional poderia dizer qual programa binário derivava de qual fonte.

Em uma trilha semelhante, quando um usuário do WINDOWS dá um clique duplo em um arquivo, um programa apropriado é carregado com o arquivo como parâmetro. O sistema operacional determina qual programa deve executar com base na extensão do arquivo.

Implementar rigidamente esses tipos de arquivos causa problemas sempre que o usuário faz qualquer coisa que os projetistas de sistema não previram. Considere, por exemplo, um sistema em que os arquivos de saída do programa têm tipo *dat* (arquivos de dados). Se um usuário escreve um formatador de programa que lê um arquivo *.pas*, transforma-o (p. ex., convertendo-o para um leiaute de alinhamento) e, então, grava o arquivo transformado como saída, o arquivo de saída será do tipo *dat*. Se o usuário tentar oferecer isso para o compilador Pascal compilá-lo, o sistema recusará porque tem a extensão errada. As tentativas de copiar *file.dat* para *file.pas* serão rejeitadas pelo sistema como inválidas (para proteger o usuário contra erros).

Embora esse tipo de "interface amigável" possa ajudar novatos, ele coloca os usuários experientes contra a parede porque eles precisam dedicar um esforço considerável para contornar a idéia que o sistema operacional tem sobre o que é razoável e o que não é.

5.1.4 Acesso a Arquivos

Os sistemas operacionais antigos ofereciam somente um tipo de acesso a arquivos: **acesso seqüencial**. Nesses sistemas, um processo poderia ler todos os bytes ou registros de um arquivo em ordem, iniciando no começo, mas não poderia pular e lê-los fora de ordem. Arquivos seqüenciais podem ser retrocedidos, entretanto, podendo, então, ser lidos conforme for necessário. Arquivos seqüenciais são convenientes quando a mídia de armazenamento é fita magnética, em vez de disco.

Quando se começou a utilizar discos para armazenar arquivos, tornou-se possível ler os bytes, ou registros de um arquivo, fora da ordem ou acessar registros por chave, em vez de por posição. Os arquivos cujos bytes ou registros podem ser lidos em qualquer ordem são chamados **arquivos de acesso aleatório**.

Arquivos de acesso aleatório são essenciais para muitos aplicativos como, por exemplo, sistemas de banco de dados. Se um cliente de linha aérea telefonar e quiser reservar um assento em um determinado vôo, o programa de reserva deve ser capaz de acessar o registro desse vôo sem primeiro ler os registros de milhares de outros vôos.

Dois métodos são utilizados para especificar onde iniciar a leitura. No primeiro, cada operação READ dá a posição no arquivo em que deve iniciar a leitura. No segundo, uma operação especial, SEEK, é oferecida para configurar a posição atual. Depois de um SEEK, o arquivo pode ser lido seqüencialmente a partir da posição atual.

Em alguns antigos sistemas operacionais de *mainframes*, os arquivos são classificados como tendo acesso seqüencial ou aleatório no momento em que eles são criados. Isso permite que o sistema utilize diferentes técnicas de armazenamento para as duas classes. Sistemas operacionais modernos não fazem essa distinção. Todos os seus arquivos são automaticamente de acesso aleatório.

5.1.5 Atributos de Arquivos

Cada arquivo tem um nome e dados. Além disso, todos os sistemas operacionais associam outras informações com cada arquivo, por exemplo, a data e a hora em que o arquivo foi criado e o tamanho do arquivo. Chamaremos esses itens extras de **atributos** do arquivo. A lista de atributos varia consideravelmente de sistema para sistema. A tabela da Figura 5-4 mostra algumas possibilidades, mas também existem outras. Nenhum sistema existente tem todas essas possibilidades, mas cada uma delas está presente em algum sistema.

Os primeiros quatro atributos relacionam-se com a proteção do arquivo e informam quem pode e quem não pode acessá-lo. Todos os tipos de esquemas são possíveis, alguns estudados mais adiante. Em alguns sistemas, o usuário deve apresentar uma senha para acessar um arquivo, caso em que a senha deve ser um dos atributos.

Os sinalizadores são bits ou campos curtos que controlam ou ativam alguma propriedade específica. Arquivos ocultos, por exemplo, não aparecem em listagens de todos os arquivos. O sinalizador de arquivo é um bit que monitora se o arquivo foi salvo em backup. O programa de backup limpa-o, e o sistema operacional configura-o sempre que um arquivo é modificado. Dessa maneira, o programa de backup pode dizer quais arquivos precisam ser salvos em backup. O sinalizador de temporário permite que um arquivo seja marcado para exclusão automática quando o processo que o criou terminar.

Os campos de comprimento do registro, de posição e de comprimento da chave somente estão presentes em arquivos cujo registros podem ser pesquisados, utilizando uma chave. Eles oferecem as informações exigidas para localizar as chaves.

Campo	Significado
Proteção	Quem pode acessar o arquivo e de que maneira
Senha	Senha necessária para acessar o arquivo
Criador	Id da pessoa que criou o arquivo
Proprietário	Proprietário atual
Sinalizador de somente-leitura	0 para leitura/gravação; 1 para somente leitura
Sinalizador de oculto	0 para normal; 1 para não exibir em listagens
Sinalizador de sistema	0 para arquivos normais; 1 para arquivo de sistema
Sinalizador de arquivo	0 para salvo em backup; 1 para ser salvo em backup
Sinalizador de ASCII/binário	0 para arquivo ASCII; 1 para arquivo binário
Sinalizador de acesso aleatório	0 para acesso seqüencial somente; 1 para acesso aleatório
Sinalizador de temporário	0 para normal; 1 para excluir o arquivo na saída do processo
Sinalizador de bloqueio	0 para destravado; não-zero para bloqueado
Comprimento do registro	Número de bytes em um registro
Posição da chave	Deslocamento da chave dentro de cada registro
Comprimento da chave	Número de bytes no campo-chave
Tempo de criação	Data e hora em que o arquivo foi criado
Tempo do último acesso	Data e hora em que o arquivo foi acessado pela última vez
Tempo da última alteração	Data e hora em que o arquivo foi alterado pela última vez
Tamanho atual	Número de bytes no arquivo
Tamanho máximo	Número de bytes até o qual o arquivo pode crescer

Figura 5-4 Alguns possíveis atributos de arquivo.

Os vários campos de tempo monitoram a data e a hora em que o arquivo foi criado, mais recentemente acessado e mais recentemente modificado. São úteis para vários propósitos. Por exemplo, um arquivo-fonte que foi modificado após a criação do arquivo-objeto correspondente precisa ser recompilado. Esses campos oferecem as informações necessárias.

O tamanho atual informa o tamanho atual do arquivo. Alguns sistemas operacionais de *mainframe* exigem que o tamanho máximo seja especificado quando o arquivo é criado, deixando o sistema operacional reservar a quantidade máxima de armazenamento de antemão. Sistemas operacionais de estações de trabalho e computadores pessoais são suficientemente inteligentes para prescindir desse recurso.

5.1.6 Operações com Arquivos

Os arquivos existem para armazenar informações e permitir que estas sejam recuperadas mais tarde. Sistemas diferentes oferecem operações diferentes para permitir armazenamento e recuperação. A seguir, discutimos as chamadas de sistema mais comuns que se relacionam com arquivos.

1. **CREATE.** O arquivo é criado sem dados. O propósito da chamada é anunciar que o arquivo está vindo e configurar alguns atributos.
2. **DELETE.** Quando o arquivo não é mais necessário, ele precisa ser excluído para liberar espaço em disco. Há sempre uma chamada de sistema para esse propósito.
3. **OPEN.** Antes de utilizar um arquivo, um processo deve abri-lo. O propósito da chamada OPEN é permitir que o sistema transfira os atributos e a lista de endereços de disco para a memória principal, permitindo acesso rápido em chamadas posteriores.
4. **CLOSE.** Quando todos os acessos terminaram, os atributos e os endereços de disco não são mais necessários, então, o arquivo deve ser fechado para liberar espaço interno na tabela. Muitos sistemas estimulam isso, impondo um número máximo de arquivos abertos em processos. Um disco é gravado em blocos, e fechar um arquivo força a gravação dos últimos blocos do arquivo, mesmo que esse bloco possa não estar ainda inteiramente cheio.
5. **READ.** Os dados são lidos do arquivo. Normalmente, os bytes provêm da posição atual. O chamador

deve especificar quantos dados são necessários e também deve oferecer um buffer onde colocá-los.

6. **WRITE.** Os dados são gravados no arquivo, novamente, em geral, na posição atual. Se a posição atual for o fim do arquivo, o tamanho do arquivo aumentará. Se a posição atual estiver no meio do arquivo, os dados existentes serão sobrescritos e perdidos para sempre.
7. **APPEND.** Essa chamada é uma forma restringida de WRITE. Ela somente pode colocar dados no fim do arquivo. Os sistemas que oferecem um conjunto mínimo de chamadas de sistema, em geral, não têm APPEND, mas muitos sistemas oferecem múltiplas maneiras de fazer a mesma coisa e esses sistemas, às vezes, a têm.
8. **SEEK.** Para arquivos de acesso aleatório, é necessário um método para especificar onde pegar os dados. Uma abordagem comum é uma chamada de sistema, SEEK, que move o ponteiro da posição atual para um lugar específico no arquivo. Depois que essa chamada completou-se, os dados podem ser lidos a partir dessa posição ou gravados nessa posição.
9. **GET ATTRIBUTES.** Os processos freqüentemente precisam ler atributos de arquivo para fazer seu trabalho. Por exemplo, o programa *make* do UNIX é comumente utilizado para gerenciar projetos de desenvolvimento de software que consistem em muitos arquivos-fonte. Quando *make* é chamado, ele examina os tempos de modificação de todos os arquivos-fonte e objeto e organiza-os de acordo com o número mínimo de compilações exigidas para atualizar tudo. Para fazer seu trabalho, ele deve olhar os atributos, especificamente nos tempos de modificação.
10. **SET ATTRIBUTES.** Alguns atributos são configuráveis pelo usuário e podem ser alterados depois que o arquivo foi criado. Essa chamada de sistema torna isso possível. As informações sobre o modo de proteção são um exemplo óbvio. A maioria dos sinalizadores também entra nessa categoria.
11. **RENAME.** Freqüentemente ocorre que um usuário precisa alterar o nome de um arquivo existente. Essa chamada de sistema torna isso possível. Ela não é sempre estritamente necessária, porque o arquivo normalmente pode ser copiado para um novo arquivo com um novo nome, e o arquivo antigo, então, é excluído.

5.2 DIRETÓRIOS

Para organizar arquivos, os sistemas de arquivos normalmente têm **diretórios** que, em muitos sistemas, também são arquivos. Nesta seção, discutiremos diretórios, sua organização, suas propriedades e as operações que neles podem ser executadas.

5.2.1 Sistemas Hierárquicos de Diretórios

Um diretório geralmente contém um certo número de entradas, uma por arquivo. Uma possibilidade é mostrada na Figura 5-5(a), na qual cada entrada contém o nome do arquivo, os atributos do arquivo e os endereços de disco onde os dados são armazenados. Outra possibilidade é mostrada na Figura 5-5(b). Aqui, uma entrada de diretório armazena o nome de arquivo e um ponteiro para outra estrutura de dados onde os atributos e os endereços de disco estão localizados. Ambos os sistemas são comumente utilizados.

Quando um arquivo é aberto, o sistema operacional pesquisa seu diretório até encontrar o nome do arquivo a ser aberto. Então, ele extrai os atributos e os endereços de disco, seja diretamente da entrada de diretório, seja da estrutura de dados para qual aponta, e coloca-os em uma tabela na memória principal. Todas as subseqüentes referências ao arquivo utilizam as informações da memória principal.

O número de diretórios varia de sistema para sistema. O projeto mais simples é o sistema manter um único diretório que contém todos os arquivos de todos os usuários,

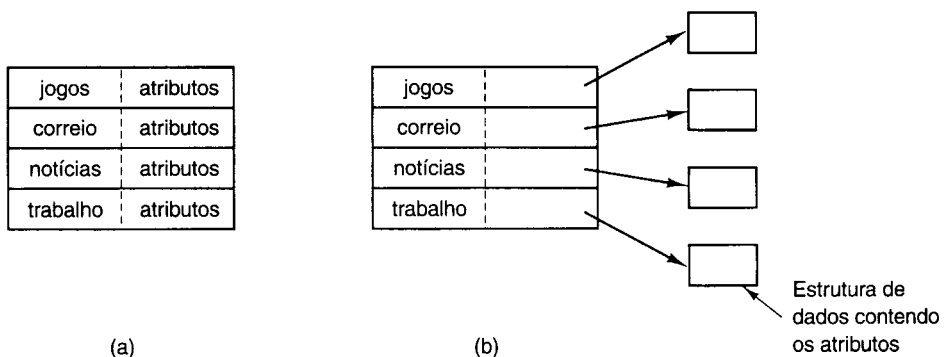



Figura 5-5 (a) Atributos na entrada de diretório. (b) Atributos em outra parte.

como ilustrado na Figura 5-6(a). Se houver muitos usuários e eles escolherem os mesmos nomes de arquivo (p. ex., *correio* e *jogos*), os conflitos e a confusão rapidamente tornarão o sistema inoperante. Esse modelo de sistema era utilizado pelo primeiro sistema operacional de microcomputador, mas raramente é visto hoje em dia.

Um aprimoramento na idéia de ter um único diretório para todos os arquivos do sistema é ter um diretório por usuário [veja Figura 5-6(b)]. Esse projeto elimina conflitos de nome entre usuários, mas não é satisfatório para usuários com um grande número de arquivos. É muito comum que usuários desejem agrupar seus arquivos de maneira lógica. Um professor, por exemplo, talvez tenha uma coleção de arquivos que, juntos, formam um livro que ele está escrevendo para um curso, uma segunda coleção de arquivos contém programas de alunos inscritos em outro curso, um terceiro grupo de arquivos contém o código de um sistema avançado de compilação que ele está construindo, um quarto grupo contém arquivos com propostas de bolsas de estudo, assim como outros arquivos para correio eletrônico, para pautas de reuniões, para papers que ele está escrevendo, para jogos e assim por diante. É necessário algum meio de agrupar tais arquivos de maneira flexível escolhida pelo usuário.

O que é necessário é uma hierarquia geral (i. e., uma árvore de diretórios). Com essa abordagem, cada usuário pode ter tantos diretórios quantos forem necessários, de modo que os arquivos podem ser agrupados de maneira natural. Essa abordagem é mostrada na Figura 5-6(c). Aqui, cada um dos diretórios *A*, *B* e *C* contidos no diretório-raiz pertence a um usuário diferente, dois dos quais criaram subdiretórios para projetos em que eles estão trabalhando.

5.2.2 Nomes de Caminho

Quando o sistema de arquivos é organizado como uma árvore de diretórios, é necessário algum meio de especificar nomes de arquivo. Dois métodos diferentes são comumente utilizados. No primeiro método, a cada arquivo é dado um **nome de caminho absoluto** que consiste no caminho do diretório-raiz até o arquivo. Como exemplo, o caminho */usr/ast/mailbox* significa que o diretório-raiz contêm um subdiretório *usr*, que, por sua vez, contém um subdiretório *ast*, que contém o arquivo *mailbox*. Nomes absolutos de caminho sempre iniciam no diretório-raiz e são únicos. No UNIX, os componentes do caminho são separados por "/". No MS-DOS, o separador é "\". No MULTICS é ">". Independentemente do caractere utilizado, se o primeiro caractere do nome de caminho for o separador, então, o caminho será absoluto.

O outro tipo de nome é o **nome de caminho relativo**. Esse é utilizado em conjunção com o conceito de **diretório de trabalho** (também chamado **diretório atual**). Um usuário pode designar um diretório como o diretório de trabalho atual, caso em que todos os nomes de caminho que não começam no diretório-raiz são interpretados em relação ao diretório de trabalho. Por exemplo, se o diretório atual de trabalho for */usr/ast*, então, o arquivo cujo caminho absoluto é */usr/ast/mailbox* pode ser referenciado simplesmente como *mailbox*. Em outras palavras, o comando UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

e o comando

```
cp mailbox mailbox.bak
```

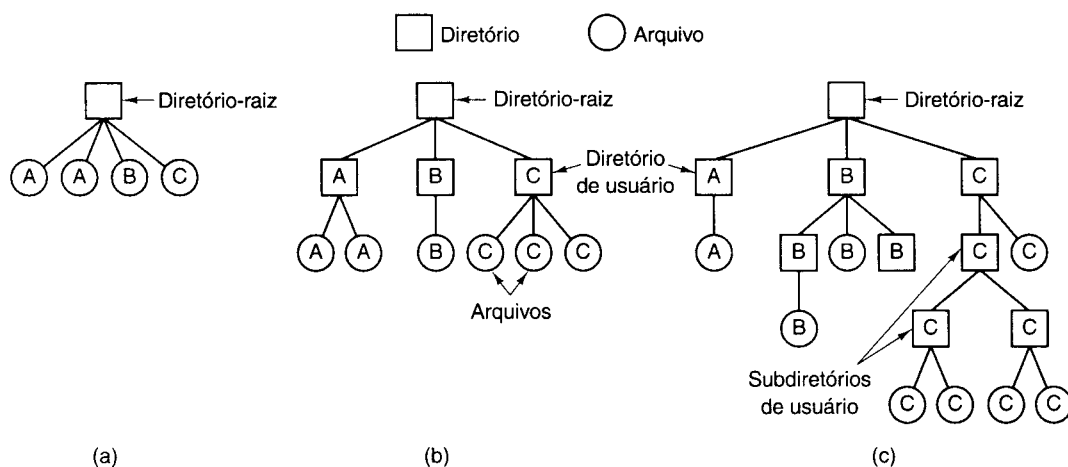



Figura 5-6 Três projetos de sistema de arquivos. (a) Único diretório compartilhado por todos usuários. (b) Um diretório por usuário. (c) Árvore arbitrária por usuário. As letras indicam o diretório ou o proprietário do arquivo.

fazem exatamente a mesma coisa se o diretório de trabalho é */usr/ast*. A forma relativa é freqüentemente mais conveniente, mas faz a mesma coisa que a forma absoluta.

Alguns programas precisam acessar um arquivo específico sem considerar qual é o diretório de trabalho. Nesse caso, eles sempre devem utilizar nomes absolutos de caminho. Por exemplo, um corretor ortográfico talvez precise ler */usr/lib/dictionary* para fazer seu trabalho. Ele deve utilizar o nome de caminho absoluto nesse caso, pois não saberá qual é o diretório de trabalho atual quando for chamado. O nome de caminho absoluto sempre funcionará, independentemente de qual seja o diretório de trabalho.

Naturalmente, se o corretor ortográfico precisar de um número grande de arquivos de */usr/lib*, uma abordagem alternativa é fazer uma chamada de sistema, alterando seu diretório de trabalho para */usr/lib* e, então, utilizar somente *dictionary* como o primeiro parâmetro para *open*. Pelo fato de alterar explicitamente o diretório de trabalho, o programa sabe com certeza em que lugar está na árvore de diretórios, portanto, ele pode utilizar caminhos relativos.

Na maioria dos sistemas, cada processo tem seu próprio diretório de trabalho, então, quando um processo altera seu diretório de trabalho e mais tarde sai, nenhum outro processo é afetado e nenhum rastro da alteração é deixado para trás no sistema de arquivos. Assim, é sempre perfeitamente seguro para um processo alterar seu diretório de trabalho sempre que for conveniente. Por outro lado, se uma biblioteca continuar alterando o diretório de trabalho e não restaurá-lo antes de terminar, o restante do programa não pode trabalhar, uma vez que sua suposição sobre onde ele está agora pode ser inválida. Por essa razão, procedimentos de biblioteca raramente alteram o diretório de trabalho e quando precisam fazer isso, sempre o restauram antes de retornar.

A maioria dos sistemas operacionais que suporta um sistema de diretórios hierárquico tem duas entradas especiais em cada diretório, "." e "..", geralmente pronunciados "ponto" e "ponto-ponto". O ponto refere-se ao diretório atual; o ponto-ponto refere-se ao seu pai. Para ver como são utilizadas, considere a árvore de arquivos UNIX da Figura 5-7. Um certo processo tem */usr/ast* como sendo seu diretório de trabalho. Ele pode utilizar ".." para subir na árvore. Por exemplo, ele pode copiar o arquivo */usr/lib/dictionary* para o próprio diretório, utilizando o comando de *shell*

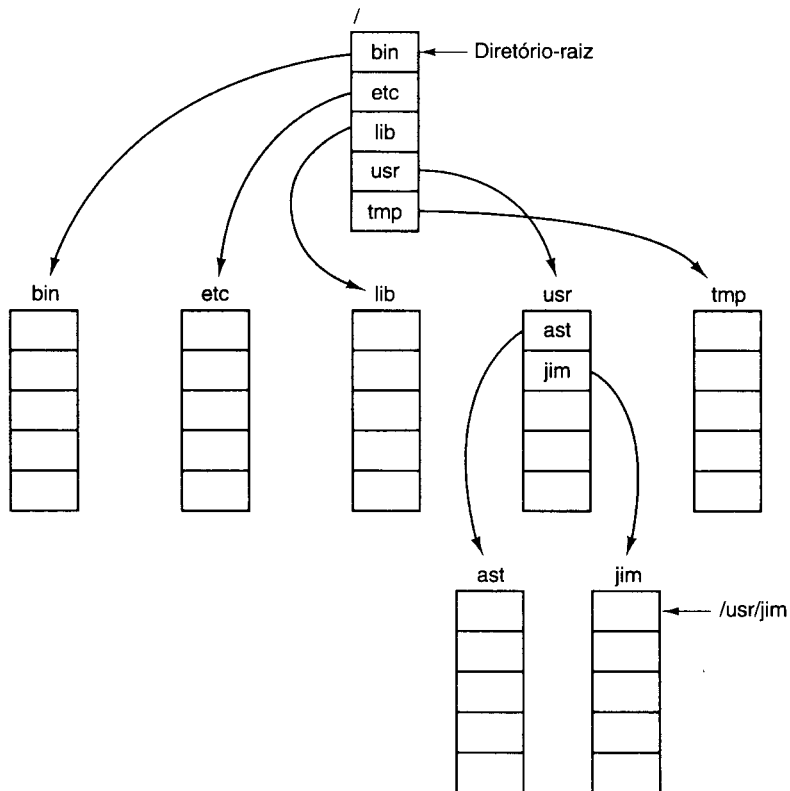



Figura 5-7 Uma árvore de diretórios UNIX.

cp ../lib/dictionary .

O primeiro caminho instrui o sistema a ir para cima (para o diretório *usr*), então, descer para o diretório *lib* para localizar o arquivo *dictionary*.

O segundo argumento dá o nome do diretório atual. Quando o comando *cp* obtém um nome de diretório (incluindo ponto) como seu último argumento, ele copia todos os arquivos aí. Naturalmente, uma maneira mais natural de fazer a cópia seria digitar

cp /usr/lib/dictionary .

aqui a utilização de pontos poupa o usuário do trabalho de digitar *dictionary* uma segunda vez.

5.2.3 Operações com Diretórios

As chamadas de sistema permitidas para gerenciar diretórios apresentam mais variação de sistema para sistema do que as chamadas de sistema para arquivos. Para dar uma idéia do que são e como funcionam, daremos um exemplo (tomado do UNIX).

1. **CREATE.** Um diretório é criado. Ele está vazio exceto por ponto e ponto-ponto, que são colocados aí automaticamente pelo sistema (ou em alguns casos, pelo programa *mkdir*).
2. **DELETE.** Um diretório é excluído. Somente um diretório vazio pode ser excluído. Um diretório que contém somente ponto e ponto-ponto é considerado vazio uma vez que esses, normalmente, não podem ser excluídos.
3. **OPENDIR.** Os diretórios podem ser lidos. Por exemplo, para listar todos os arquivos em um diretório, um programa de listagem abre o diretório para ler o nome de todos os arquivos que ele contém. Antes de um diretório poder ser lido, ele deve ser aberto de maneira análoga à abertura e à leitura de um arquivo.
4. **CLOSEDIR.** Quando um diretório foi lido, ele deve ser fechado para liberar espaço interno da tabela.
5. **READDIR.** Essa chamada retorna a próxima entrada em um diretório aberto. Antigamente, era possível ler diretórios utilizando a chamada de sistema READ normal. Mas essa abordagem tem a desvantagem de forçar o programador a conhecer e a lidar com a estrutura interna de diretórios. Ao contrário, READDIR sempre retorna uma entrada em um formato padrão, independentemente de qual das possíveis estruturas de diretório está sendo utilizada.
6. **RENAME.** Sob muitos aspectos, os diretórios são simplesmente arquivos e podem ser renomeados da mesma maneira como os arquivos.
7. **LINK.** Vinculação é uma técnica que permite que um arquivo apareça em mais de um diretório. Essa chamada de sistema especifica um arquivo existente e um nome de caminho e cria um vínculo do arquivo existente para o nome especificado pelo caminho. Assim, o mesmo arquivo pode aparecer em múltiplos diretórios.
8. **UNLINK.** Uma entrada de diretório é removida. Se o arquivo sendo desvinculado está presente em apenas um diretório (o caso normal), ele é removido do sistema de arquivos. Se estiver presente em múltiplos diretórios, somente o nome de caminho especificado é removido. Os outros permanecem. No UNIX, a chamada de sistema para excluir arquivos (discutida anteriormente) é, de fato, UNLINK.

A lista anterior fornece as chamadas mais importantes, mas há algumas outras também para, por exemplo, gerenciar as informações de proteção associadas com um diretório.

5.3 IMPLEMENTAÇÃO DO SISTEMA DE ARQUIVOS

Agora é hora de mudar do ponto de vista do usuário do sistema de arquivos para o enfoque do implementador. Os usuários estão preocupados com a maneira como os arquivos são nomeados, que operações são permitidas neles, a aparência que a árvore de diretórios tem e questões de interface. Os implementadores estão interessados em como arquivos e diretórios são armazenados, em como o espaço em disco é gerenciado e em como fazer tudo funcionar eficiente e confiavelmente. Nas seções a seguir, examinaremos algumas dessas áreas para ver quais são as questões e as opções envolvidas.

5.3.1 Implementando Arquivos

Provavelmente a questão mais importante ao implementar armazenamento de arquivos é monitorar quais blocos de disco acompanham quais arquivos. Vários métodos são utilizados em diferentes sistemas operacionais. Nesta seção, examinaremos alguns deles.

Alocação Contígua

O esquema mais simples de alocação é armazenar cada arquivo como um bloco contíguo de dados no disco. Assim, em um disco com blocos de 1K, um arquivo de 50K alocaria 50 blocos consecutivos. Esse esquema tem duas vantagens significativas. Em primeiro lugar, é simples de implementar porque monitorar onde os blocos de um arquivo estão reduz-se a lembrar um número, o endereço de disco do primeiro bloco. Em segundo lugar, o desempenho é excelente porque o arquivo inteiro pode ser lido do disco em uma única operação.

A alocação contígua, infelizmente, também tem duas desvantagens igualmente significativas. Primeiro, não é praticável a menos que o tamanho máximo do arquivo seja conhecido no momento em que o arquivo é criado. Sem essa informação, o sistema operacional não sabe quanto espaço em disco reservar. Entretanto, em sistemas em que os arquivos devem ser gravados de uma única tacada, ela pode ser utilizada com grande vantagem.

A segunda desvantagem é a fragmentação do disco que resulta dessa política de alocação. É desperdiçado espaço que, de outra maneira, talvez pudesse ser utilizado. A compactação de disco normalmente tem um custo proibitivo, embora concebivelmente possa ser feita à noite, quando o sistema está desocupado.

Alocação por Lista Encadeada

O segundo método para armazenar arquivos é manter cada um como uma lista encadeada de blocos de disco, como mostrado na Figura 5-8. A primeira palavra de cada bloco é utilizada como um ponteiro para o seguinte. O resto do bloco é para dados.

Diferentemente da alocação contígua, todos os blocos do disco podem ser utilizados nesse método. Nenhum espaço é desperdiçado em fragmentação de disco (exceto em fragmentação interna no último bloco). Além disso, é suficiente para a entrada de diretório meramente armazenar o endereço de disco do primeiro bloco. O restante pode ser encontrado iniciando aí.

Por outro lado, embora a leitura seqüencial de um arquivo seja simples e direta, o acesso aleatório é extremamente lento. Além disso, o espaço de dados em um bloco não é mais uma potência de dois porque o ponteiro ocupa alguns bytes. Embora não fatal, ter um tamanho peculiar é menos eficiente porque muitos programas lêem e gravam em blocos cujo tamanho é uma potência de dois.

Alocação por Lista Encadeada Utilizando um Índice

As duas desvantagens da alocação por lista encadeada podem ser eliminadas pegando a palavra de ponteiro de cada bloco de disco e colocando-a em uma tabela ou em um índice na memória. A Figura 5-9 mostra a aparência que tem a tabela para o exemplo da Figura 5-8. Em ambas as figuras, temos dois arquivos. O arquivo *A* utiliza os blocos de disco 4, 7, 2, 10 e 12, nessa ordem, e o arquivo *B* utiliza os blocos de disco 6, 3, 11 e 14, nessa ordem. Utilizando a tabela da Figura 5-9, podemos iniciar com o bloco 4 e seguir a cadeia completamente até o fim. O mesmo pode ser feito iniciando com o bloco 6.

Utilizando essa organização, o bloco inteiro está disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Embora a cadeia ainda deva ser seguida para localizar-se um dado deslocamento dentro do arquivo, a cadeia está inteiramente na memória, portanto, ela pode ser seguida sem fazer qualquer referência de disco. Como no método anterior, é suficiente que a entrada de diretório mantenha um inteiro simples (o número do bloco inicial) e ainda seja capaz de localizar todos os blocos, independentemente do tamanho do arquivo. O MS-DOS utiliza esse método para alocação de disco.

A desvantagem principal desse método é que a tabela inteira deve estar na memória todo o tempo para fazê-lo funcionar. Com um disco grande, digamos, 500.000 blocos de 1K (500 M), a tabela terá 500.000 entradas, cada uma da quais com o mínimo 3 bytes. Para acelerar as consultas, elas devem ser de 4 bytes. Assim, a tabela ocupará 1,5 ou 2 megabytes todo o tempo dependendo de o sistema ser otimizado para espaço ou tempo. Embora o MS-DOS utilize esse mecanismo, ele evita tabelas enormes, utilizando blocos grandes (até 32K) em discos grandes.

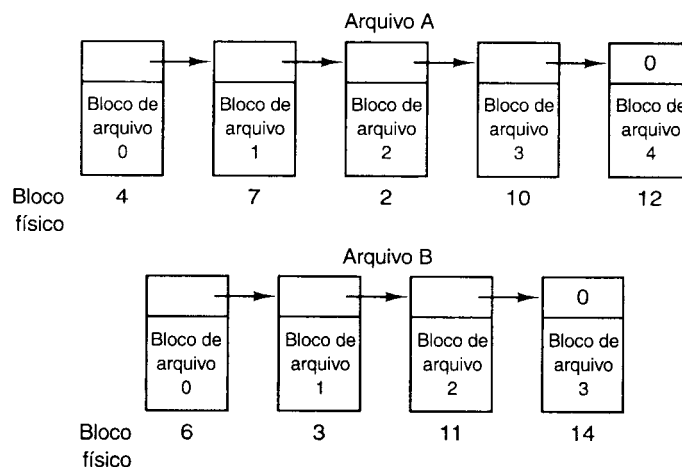



Figura 5-8 Armazenando um arquivo como uma lista encadeada de blocos de disco.

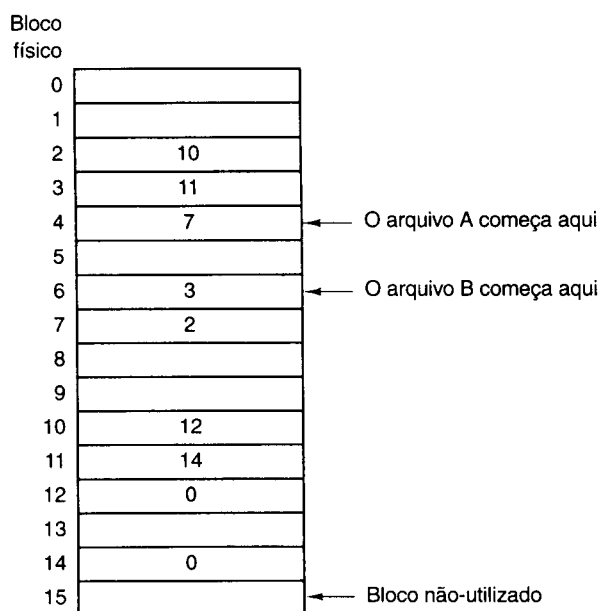



Figura 5-9 Alocação por lista encadeada, utilizando uma tabela na memória principal.

Nós-I

Nosso último método para monitorar quais blocos pertencem a quais arquivos é associar com cada arquivo uma pequena tabela chamada **nó-i (nó de índice)**, que lista os atributos e os endereços de disco dos blocos do arquivo, como mostrado na Figura 5-10.

Os primeiros endereços de disco são armazenados no próprio nó-i, então, para arquivos pequenos, todas as informações necessárias estão diretamente no nó-i, que é carregado do disco para a memória principal quando o arquivo é aberto. Para arquivos maiores, um dos endereços no nó-i é o endereço de um bloco de disco chamado **bloco indireto simples**. Esse bloco contém endereços adicionais de disco. Se isso ainda não for suficiente, outro endereço no nó-i, chamado **bloco indireto duplo**, contém o endereço de um bloco que, por sua vez, contém uma lista de blocos indiretos simples. Cada um desses blocos indiretos simples aponta para algumas centenas de blocos de dados. Se isso ainda não for suficiente, um **bloco indireto triplo** também poderá ser utilizado. O UNIX utiliza esse esquema.

5.3.2 Implementando Diretórios

Antes de um arquivo poder ser lido, ele deve ser aberto. Quando um arquivo é aberto, o sistema operacional utiliza o nome de caminho fornecido pelo usuário para localizar a entrada de diretório. A entrada de diretório oferece as informações necessárias para localizar os blocos de disco. Dependendo do sistema, tais informações podem ser o endereço de disco do arquivo inteiro (alocação contígua), o número do primeiro bloco (ambos os esquemas de lista encadeada) ou o número do nó-i. Em todos os casos, a principal função do sistema de diretórios é mapear o nome ASCII do arquivo para as informações necessárias para localizar os dados.

Uma questão intimamente relacionada é onde os atributos devem ser armazenados. Uma possibilidade óbvia é armazená-los diretamente na entrada de diretório. Muitos sistemas fazem exatamente isso. Para sistemas utilizando nós-i, outra possibilidade é armazenar os atributos no nó-i, em vez de na entrada de diretório. Como veremos mais tarde, esse método tem certas vantagens sobre colocar os atributos na entrada de diretório.

Diretórios no CP/M

Vamos iniciar nosso estudo de diretórios com um exemplo particularmente simples, o do CP/M (Golden e Pechura, 1986), ilustrado na Figura 5-11. Nesse sistema, há somente um diretório, portanto, tudo o que o sistema de arquivos precisa fazer para procurar um nome de arquivo é pesquisar exclusivamente esse diretório. Quando localiza a entrada, ele também tem o número de blocos de disco, uma vez que eles estão armazenados logo na entrada de diretório, como também todos os atributos. Se um arquivo utiliza mais blocos de disco do que os que se ajustam em uma entrada, o arquivo aloca entradas de diretório adicionais.

Os campos na Figura 5-11 têm os seguintes significados. O campo *código do usuário* monitora qual usuário é proprietário do arquivo. Durante uma pesquisa, somente

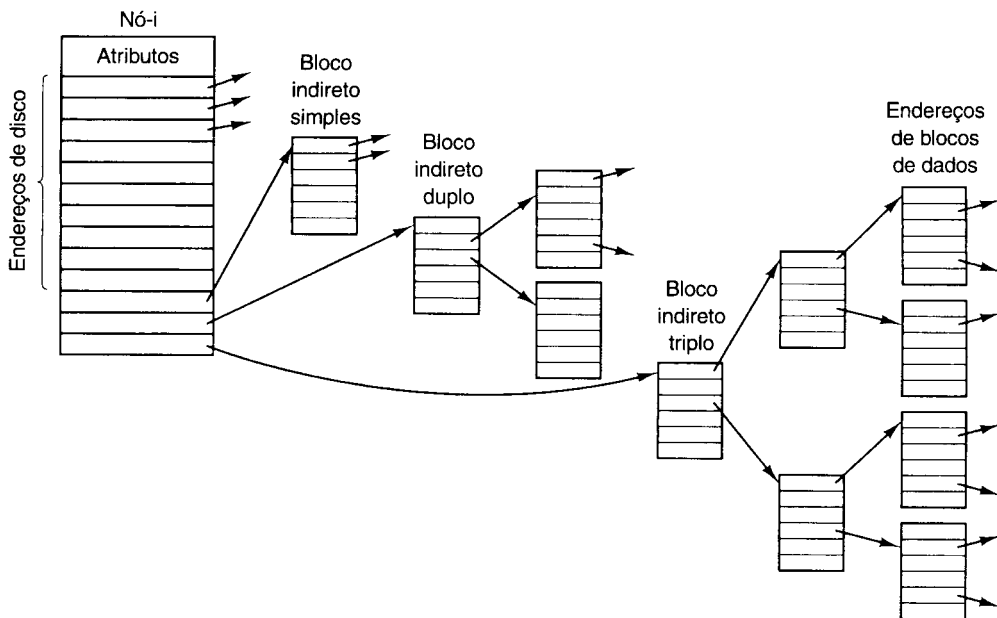



Figura 5-10 Um nó-i.

as entradas que pertencem ao usuário atualmente conectado são verificadas. Os próximos dois campos dão o nome e a extensão do arquivo. O campo *grau* é necessário porque um arquivo maior que 16 blocos ocupa múltiplas entradas de diretório. Esse campo é utilizado para dizer qual entrada vem em primeiro, em segundo lugar e assim por diante. O campo *contagem de blocos* diz quantas das 16 possíveis entradas de bloco de disco estão em utilização. Os 16 campos finais contêm os próprios números de bloco de disco. O último bloco pode não estar cheio, então, o sistema não tem como determinar o tamanho exato de um arquivo até o último byte (i. e., ele monitora tamanhos de arquivo em blocos, não em bytes).

Diretórios no MS-DOS

Agora vamos considerar alguns exemplos de sistemas com árvores de diretório hierárquicas. A Figura 5-12 mostra uma entrada de diretório do MS-DOS. Ela tem 32 bytes de comprimento e contém o nome do arquivo, os atributos e o número do primeiro bloco de disco. O número do primeiro bloco é utilizado como um índice em uma tabela do tipo da Figura 5-9. Seguindo a cadeia, todos os blocos podem ser localizados.

No MS-DOS, os diretórios podem conter outros diretórios, conduzindo a um sistema de arquivos hierárquico. É comum no MS-DOS que diferentes programas aplicativos iniciem criando um diretório no diretório raiz e colocando todos seus arquivos aí, para não haver conflito com aplicativos diferentes.

Diretórios no UNIX

A estrutura de diretórios tradicionalmente utilizada no UNIX é extremamente simples, como mostrado na Figura 5-13. Cada entrada contém apenas um nome de arquivo e o número de seu nó-i. Todas as informações sobre o tipo, tamanho, tempos, proprietário e blocos de disco estão con-

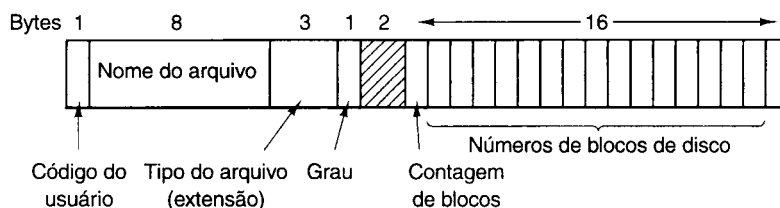



Figura 5-11 Uma entrada de diretório que contém os números de bloco de disco para cada arquivo.

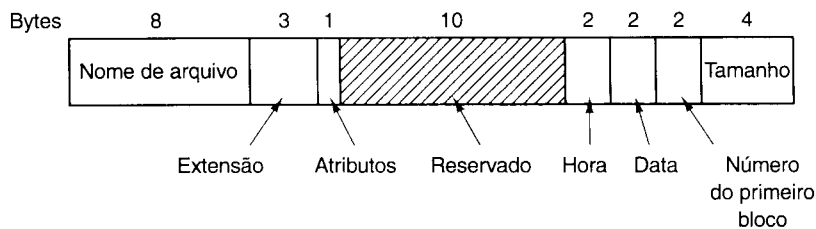



Figura 5-12 Entrada de diretório no MS-DOS.

tidas no nó-i. Alguns sistemas UNIX têm um arranjo diferente, mas em todos os casos, uma entrada de diretório contém em última instância apenas uma *string* ASCII e um número de nó-i.

Quando um arquivo é aberto, o sistema de arquivos deve pegar o nome de arquivo fornecido e localizar seus blocos de disco. Vamos considerar como o nome de caminho */usr/ast/mbox* é localizado. Utilizaremos UNIX como um exemplo, mas o algoritmo é basicamente o mesmo para todos os sistemas de diretório hierárquicos. Primeiro o sistema de arquivos localiza o diretório-raiz. No UNIX, seu nó-i está localizado em um lugar fixo no disco.

Então, o sistema de arquivos busca o primeiro componente do caminho, *usr*, no diretório-raiz para localizar o úmero de nó-i do arquivo */usr*. Localizar um nó-i a partir de seu número é simples e direto, uma vez que cada um tem uma posição fixa no disco. A partir desse nó-i, o sistema localiza o diretório para */usr* e pesquisa nele o próximo componente, *ast*. Quando encontrar a entrada para *ast*, o sistema terá o nó-i para o diretório */usr/ast*. A partir desse nó-i, o sistema pode localizar o diretório em si e pesquisa *mbox*. O nó-i para esse arquivo, então, é lido na memória e mantido aí até o arquivo ser fechado. O processo de pesquisa é ilustrado na Figura 5-14.

Nomes de caminho relativos são pesquisados da mesma maneira que os absolutos, exceto que iniciam a partir do diretório de trabalho em vez de iniciar a partir do diretório-raiz. Cada diretório tem entradas para "." e ".." que são colocados aí quando o diretório é criado. A entrada "." tem o número de nó-i para o diretório atual, e a entrada para ".." tem o número de nó-i para o diretório-pai. Assim, um procedimento procurando por *../dick/prog.c* simplesmente pesquisa ".." no diretório de trabalho, localiza o número de nó-i do diretório-pai e busca por *dick* nesse diretório. Nenhum mecanismo especial é necessário para tratar esses nomes. No que diz respeito ao sistema de diretório, eles são simplesmente cadeias de caracteres ASCII, como quaisquer outros nomes.

5.3.3 Gerenciamento de Espaço em Disco

Os arquivos normalmente são armazenados em disco, portanto, o gerenciamento de espaço em disco é uma questão importante para projetistas de sistema de arquivos. Duas estratégias gerais são possíveis para armazenar um arquivo de n bytes: os n bytes consecutivos de espaço em disco são alocados ou o arquivo é dividido em um número de blocos (não necessariamente) contíguos. As mesmas escolhas devem ser feitas em sistemas de gerenciamento de memória entre segmentação pura e paginação.

Armazenar um arquivo como uma seqüência contígua de bytes tem o problema óbvio de que, se um arquivo crescer, ele provavelmente precisará ser movido no disco. O mesmo problema aplica-se a segmentos na memória, exceto que mover um segmento na memória é uma operação relativamente rápida se comparada com mover um arquivo de uma posição no disco para outra. Por essa razão, quase todos os sistemas dividem os arquivos em blocos de tamanho fixo que não precisam ser adjacentes.

Tamanho de Bloco

Uma vez que foi decidido armazenar arquivos em blocos de tamanho fixo, surge a pergunta de qual tamanho o bloco deve ter. Dada a maneira como os discos são organi-

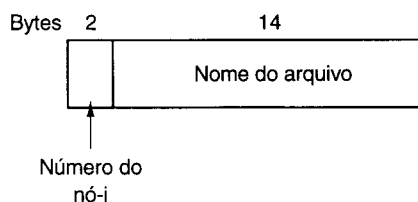



Figura 5-13 Uma entrada de diretório no UNIX.

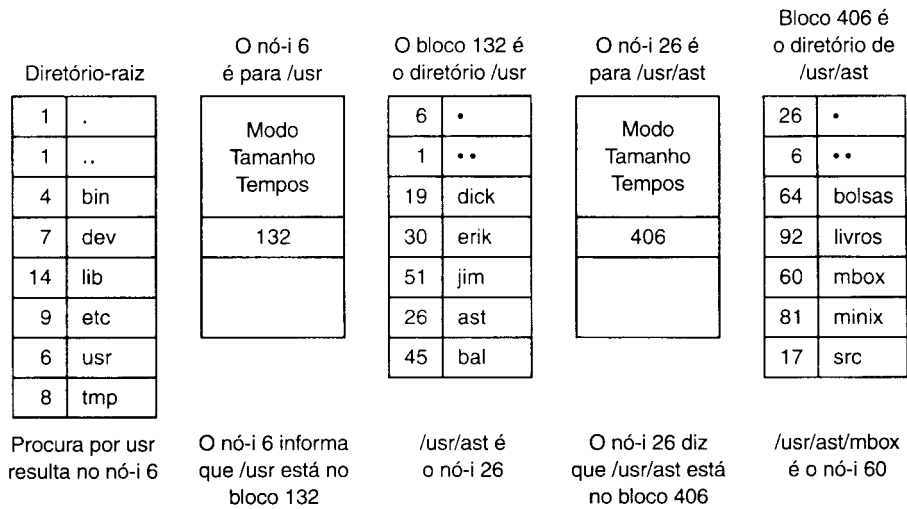



Figura 5-14 Os passos ao pesquisar */usr/ast/mbox*.

zados, o setor, a trilha e o cilindro são candidatos óbvios para a unidade de alocação. Em um sistema com paginação, o tamanho da página é também um competidor importante.

Ter uma unidade grande de alocação, como um cilindro, significa que cada arquivo, mesmo um arquivo de 1 byte, ocupa um cilindro inteiro. Estudos (Mullender e Tanenbaum, 1984) demostraram que o tamanho médio de arquivo em ambientes UNIX está por volta de 1K, então, alocar um cilindro de 32K para cada arquivo desperdiçaria 31/32 ou 97% do espaço total em disco. Por outro lado, utilizar uma unidade de alocação pequena significa que cada arquivo consistirá em muitos blocos. A leitura de cada bloco normalmente exige uma busca e um atraso rotacional, portanto, ler um arquivo consistindo em muitos blocos pequenos é lento.

Como um exemplo, considere um disco com 32.768 bytes por trilha, um tempo de rotação de 16,67ms e um tempo de busca médio de 30ms. O tempo em milissegundos para ler um bloco de k bytes é, então, a soma dos tempos de busca, de atraso rotacional e transferência:

$$30 + 8{,}3 + (k / 32768) \times 16{,}67$$

A curva sólida da Figura 5-15 mostra a taxa de dados para tal disco como uma função do tamanho de bloco. Se fizermos a suposição grosseira de que todos os arquivos têm 1K (o tamanho médio medido), a curva tracejada dessa figura dá a eficiência do espaço em disco. A má notícia é que a boa utilização de espaço (tamanho de bloco < 2K) significa taxas de dados baixas e vice-versa. A eficiência de tempo e de espaço estão inerentemente em conflito.

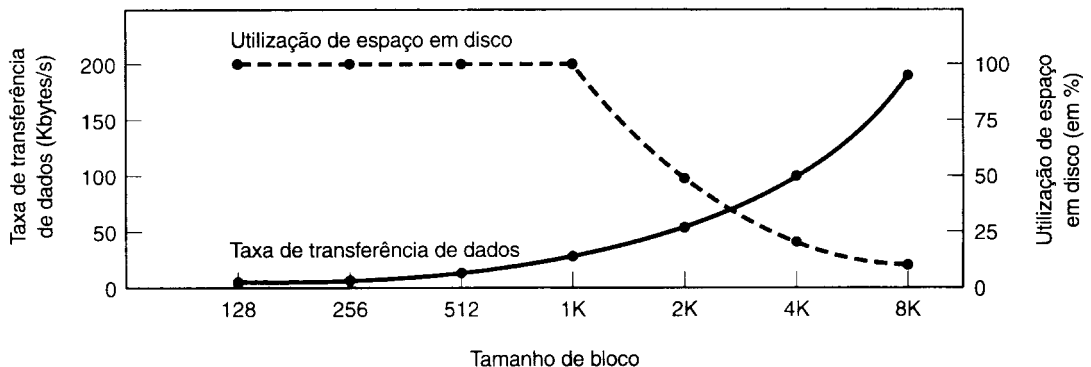



Figura 5-15 A curva sólida (escala da esquerda) fornece a taxa de transferência de dados de um disco. A curva tracejada (escala da direita) fornece a eficiência do espaço em disco. Todos os arquivos são de 1K.

O ajuste normal é escolher um tamanho de bloco de 512, 1K ou 2K bytes. Se um tamanho de bloco de 1K for escolhido em um disco com um tamanho de setor de 512 bytes, então, o sistema de arquivos sempre lerá ou gravará dois setores consecutivos e irá tratá-los como uma única unidade indivisível. Qualquer que seja a decisão tomada, provavelmente ela deve ser reavaliada periodicamente, uma vez que, como com todos aspectos da tecnologia de computador, os usuários tiram proveito dos recursos mais abundantes, exigindo cada vez mais. Um gerenciador de sistema informa que o tamanho médio de arquivos no sistema de uma universidade que ele gerencia aumentou lentamente com os anos e que em 1997, o tamanho médio de arquivos cresceu para 12K, no caso dos alunos, e para 15K, no caso dos profissionais e dos professores da faculdade.

Monitorando Blocos Livres

Uma vez que um tamanho de bloco foi escolhido, a próxima questão é como monitorar blocos livres. Dois métodos são amplamente utilizados, como mostrado na Figura 5-16. O primeiro consiste em utilizar uma lista encadeada de blocos de disco, com cada bloco armazenando tantos quantos números livres de bloco de disco couberem. Com um bloco de 1K e um número de bloco de disco de 32 bits, cada bloco na lista de livres armazena os números de 255 blocos livres. (Uma entrada é necessária para o ponteiro para o próximo bloco). Um disco de 200MB necessita de uma lista de livres de no máximo 804 blocos para armazenar todos os 200K números de bloco de disco. Blocos livres freqüentemente são utilizados para armazenar a lista de livres.

A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos requer um mapa de bits com n bits. Blocos livres são representados por 1s no mapa, e blocos alocados por 0s (ou vice-versa). Um disco de 200MB requer 200K bits para o mapa, o que requer somente 25 blocos. Não é de surpreender que o mapa de bits exija menos espaço, uma vez que utiliza 1 bit por bloco, *versus* 32 bits no modelo de lista encadeada. Somente se o disco estiver quase cheio é que o esquema de lista encadeada irá requerer menos blocos que o mapa de bits.

Se houver memória principal suficiente para armazenar o mapa de bits, esse método é geralmente preferível. Se, entretanto, somente 1 bloco de memória puder ser dispensado para monitorar blocos livres no disco e o disco estiver quase cheio, então, a lista encadeada pode ser melhor. Com somente 1 bloco de mapa de bits na memória, é possível que nenhum bloco livre possa ser encontrado, fazendo com que acessos de disco adicionais sejam necessários para ler o restante do mapa de bits. Quando um bloco novo da lista encadeada é carregado na memória, 255 blocos de disco podem ser alocados antes de ir-se para o disco a fim de buscar o próximo bloco da lista.

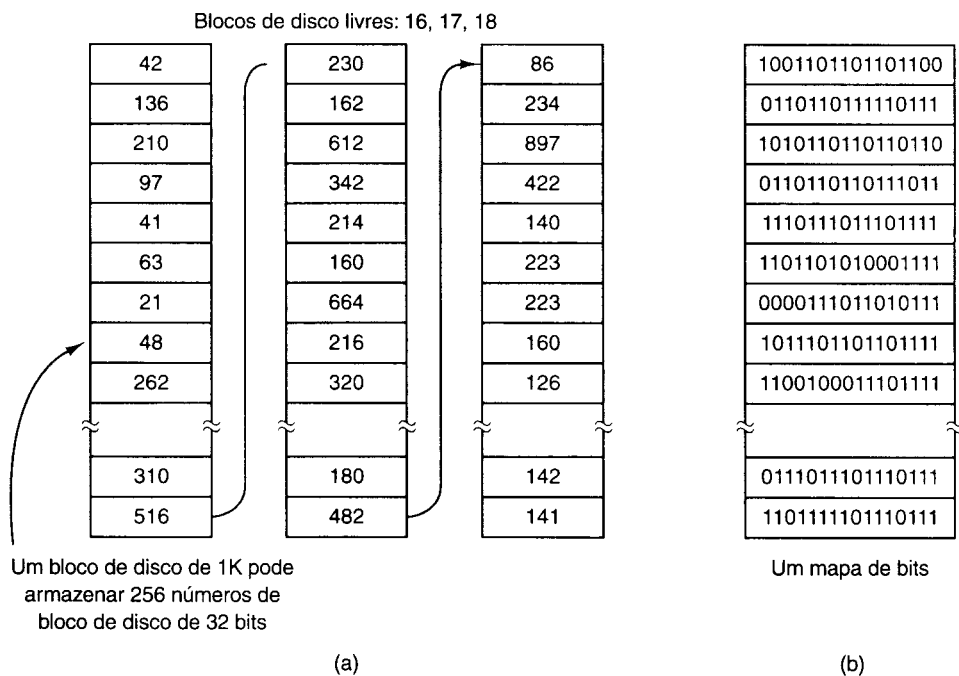



Figura 5-16 (a) Armazenando a lista de livres em uma lista encadeada. (b) Um mapa de bits.

5.3.4 Confiabilidade do Sistema de Arquivos

A destruição de um sistema de arquivos é freqüentemente um desastre muito maior do que a destruição de um computador inteiro. Se um computador é destruído por fogo, oscilações em virtude de relâmpagos ou uma xícara de café derramada sobre o teclado, isso é irritante e custará dinheiro, mas geralmente uma peça de reposição pode ser adquirida com um mínimo de problemas. Computadores pessoais baratos podem até ser substituídos dentro de algumas horas com uma simples visita à revenda de produtos de informática mais próxima (exceto nas universidades, onde fazer um pedido de compra exige três comitês, cinco assinaturas e 90 dias).

Se um sistema de arquivos do computador é irrevogavelmente perdido, seja devido a hardware, software ou a ratos que roeram os disquetes, restaurar todas as informações será difícil, consumirá tempo e, em muitos casos, será impossível. Para pessoas cujos programas, documentos, arquivos de clientes, registros de impostos, bancos de dados, planos de marketing ou outros dados perderam-se para sempre, as conseqüências podem ser catastróficas. Embora o sistema de arquivos não possa oferecer qualquer proteção contra destruição física do equipamento e da mídia, ele pode ajudar a proteger as informações. Nesta seção, veremos algumas questões envolvidas na salvaguarda do sistema de arquivos.

Os discos podem ter blocos defeituosos, como indicamos no Capítulo 3. Os disquetes geralmente estão perfeitos quando deixam a fábrica, mas podem desenvolver blocos defeituosos durante a utilização. Os discos winchester freqüentemente têm blocos defeituosos logo de início: é simplesmente muito caro fabricá-los completamente livres de todos os defeitos. De fato, discos rígidos mais antigos costumavam ser fornecidos com uma lista dos blocos defeituosos descobertos pelos testes do fabricante. Nesses discos, um setor é reservado para uma lista de blocos defeituosos. Quando a controladora é inicializada pela primeira vez, ela lê a lista de blocos defeituosos e seleciona um bloco (ou trilha) sobressalente para substituir os defeituosos, registrando o mapeamento na lista de blocos defeituosos. Daí em diante, todas as solicitações para o bloco defeituoso utilizarão o sobressalente. Quando novos erros forem descobertos, essa lista é atualizada como parte de uma formatação de baixo nível.

Houve uma melhora constante nas técnicas de fabricação, assim os blocos defeituosos são menos comuns que antigamente. Entretanto, eles ainda ocorrem. A controladora em uma unidade moderna de disco é muito sofisticada, como observado no Capítulo 3. Nesses discos, as trilhas têm pelo menos um setor a mais que o necessário, de modo que pelo menos um trecho defeituoso pode ser pulado, deixando-o em uma lacuna entre dois setores consecutivos. Há também alguns setores sobressalentes por cilindro, portanto, a controladora pode fazer remapeamento automático do setor se notar que um setor precisa de mais que um certo número de tentativas para ser lido ou gravado. Assim, o usuário normalmente não está ciente dos blocos defeituosos ou de seu gerenciamento. Contudo, quando uma IDE moderna ou disco de SCSI falha, normalmente falhará de maneira catastrófica, porque os setores sobressalentes esgotam-se. Os discos SCSI oferecem um aviso de "erro recuperado" ao remapear um bloco. Se o *driver* perceber isso e imprimir uma mensagem no console, o usuário saberá que é hora de comprar um novo disco quando essas mensagens começarem a aparecer com freqüência.

Há uma solução simples de software para o problema de bloco defeituoso, adequada para utilização em discos mais antigos. Essa abordagem requer que o usuário ou o sistema de arquivos construa com cuidado um arquivo que contenha todos os blocos defeituosos. Essa técnica remove-os da lista de livres, então, eles nunca ocorrerão em arquivos de dados. Contanto que o arquivo de blocos defeituosos nunca seja lido ou gravado, nenhum problema surgirá. Cuidado precisa ser tomado durante backups de disco para evitar a leitura desse arquivo.

Backups

Mesmo com uma estratégia inteligente para lidar com blocos defeituosos, é importante fazer backup dos arquivos freqüentemente. Afinal de contas, alternar automaticamente para uma trilha sobressalente depois que blocos de dados cruciais foram arruinados é algo parecido como fechar a porta do celeiro depois que o cavalo de raça premiado escapou.

Os sistemas de arquivos em disquete podem ser salvos em backup simplesmente copiando o disquete inteiro para um em branco. Os sistemas de arquivos em discos winchester pequenos podem ser salvos em backup fazendo cópia do disco inteiro para fita magnética. Tecnologias atuais incluem cartuchos de fita de 150 M e fitas *Exabyte* ou DAT de 8 G.

Para winchesters grandes (p. ex., de 10 GB), fazer backup da unidade inteira em fita é um incômodo e consome tempo. Uma estratégia que é fácil de implementar, mas desperdiça metade do armazenamento é oferecer a cada computador duas unidades em vez de uma. Ambas as unidades são divididas em duas metades: dados e backup. Cada noite, a parte de dados da unidade 0 é copiada para a parte de backup da unidade 1 e vice-versa, como mostrado na Figura 5-17. Dessa maneira, se uma unidade for completamente arruinada, nenhuma informação será perdida.

Uma alternativa para fazer a cópia do sistema de arquivos inteiro todo dia são as **cópias incrementais**. A forma simples de fazer uma cópia incremental é fazer uma cópia completa periodicamente, digamos, semanalmente ou mensalmente, e fazer uma cópia diária somente dos arquivos que foram modificados desde a última cópia completa. Um esquema melhor é copiar somente os arquivos alterados desde que eles foram copiados pela última vez.

Para implementar esse método, uma lista dos tempos de cópia para cada arquivo deve ser mantida no disco. O

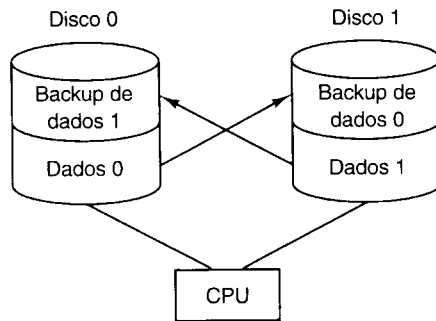



Figura 5-17 Fazer backup de uma unidade em outra desperdiça metade do espaço de armazenamento.

programa de cópia, então, verifica cada arquivo no disco. Se esse foi modificado desde a última cópia, ele é copiado novamente e seu tempo de última cópia é alterado para o tempo atual. Se feito em um ciclo mensal, esse método requer 31 fitas diárias de cópia, uma por dia, mais as fitas suficientes para armazenar uma cópia completa, feita uma vez por mês. Outros esquemas mais complexos que utilizam menos fitas também estão em utilização.

Métodos automáticos que utilizam múltiplos discos também são utilizados. Por exemplo, o **espelhamento** utiliza dois discos. Gravações vão para ambos os discos, e leituras vêm de um. A gravação para o disco de espelho é atrasada um pouco, de modo que possa ser feita quando o sistema esteja desocupado. Um sistema assim pode continuar a executar em "modo degradado" quando um disco falha, permitindo que um disco defeituoso seja substituído, e os dados sejam recuperados sem parada do sistema.

Consistência do Sistema de Arquivos

Outra área onde a confiabilidade é uma questão é a consistência do sistema de arquivos. Muitos sistemas de arquivos lêem blocos, modificam-nos e gravam-nos mais tarde. Se o sistema cai antes de todos os blocos modificados serem gravados, o sistema de arquivos pode ser deixado em um estado inconsistente. Esse problema é especialmente crítico se alguns blocos que não foram gravados forem blocos de nó-i, blocos de diretório ou blocos contendo a lista de livres.

Para lidar com o problema de sistemas de arquivos inconsistentes, a maioria dos computadores tem um programa utilitário que verifica a consistência do sistema de arquivos. Ele pode ser executado sempre que o sistema é inicializado, especialmente depois de uma queda. A descrição a seguir diz como esse utilitário funciona no UNIX e no MINIX; outros sistemas têm algo semelhante. Esses verificadores de sistema de arquivos verificam cada sistema de arquivos (disco) independentemente dos demais.

Dois tipos de verificação de consistência podem ser feitos: blocos e arquivos. Para verificar consistência de blocos, o programa constrói duas tabelas, cada uma contendo um contador para cada bloco, inicialmente configurado como 0. Os contadores na primeira tabela monitoram quantas vezes cada bloco está presente em um arquivo; os contadores na segunda tabela registram a freqüência com que cada bloco está presente na lista de livres (ou no mapa de bits de blocos livres)

O programa, então, lê todos os nós-i. Iniciando de um nó-i, é possível construir uma lista de todos os números de bloco utilizados no arquivo correspondente. À medida que cada número de bloco é lido, seu contador na primeira tabela é incrementado. O programa, então, examina a lista de livres ou de mapa de bits, para localizar todos os blocos que não estão em utilização. Cada ocorrência de um bloco na lista de livres resulta no incremento do seu contador na segunda tabela.

Se o sistema de arquivos é consistente, cada bloco terá 1 na primeira tabela ou na segunda tabela, como ilustrado na Figura 5-18(a). Entretanto, como um resultado de uma queda, as tabelas talvez se pareçam com a Figura 5-18(b), na qual os blocos 2 não ocorrem em nenhuma tabela. Ele será informado como sendo um **bloco ausente**. Embora os blocos ausentes não causem nenhum dano real, eles desperdiçam espaço e, portanto, reduzem a capacidade do disco. A solução para blocos ausentes é simples e direta: o verificador do sistema de arquivos simplesmente adiciona-os à lista livre.

Outra situação que talvez ocorra é a da Figura 5-18(c). Aqui vemos um bloco, número 4, que ocorre duas vezes na lista de livres. (Duplicatas podem ocorrer somente se a lista de livres for realmente uma lista; com um mapa de bits é impossível.) A solução aqui também é simples: reconstruir a lista de livres.

A pior coisa que pode acontecer é o mesmo bloco de dados estar presente em dois ou mais arquivos, como mostrado na Figura 5-18(d) com o bloco 5. Se qualquer um desses arquivos for removido, o bloco 5 será colocado na lista de livres, o que leva a uma situação em que o mesmo bloco está tanto em utilização como livre ao mesmo tempo. Se os dois arquivos forem removidos, o bloco será colocado na lista de livres duas vezes.

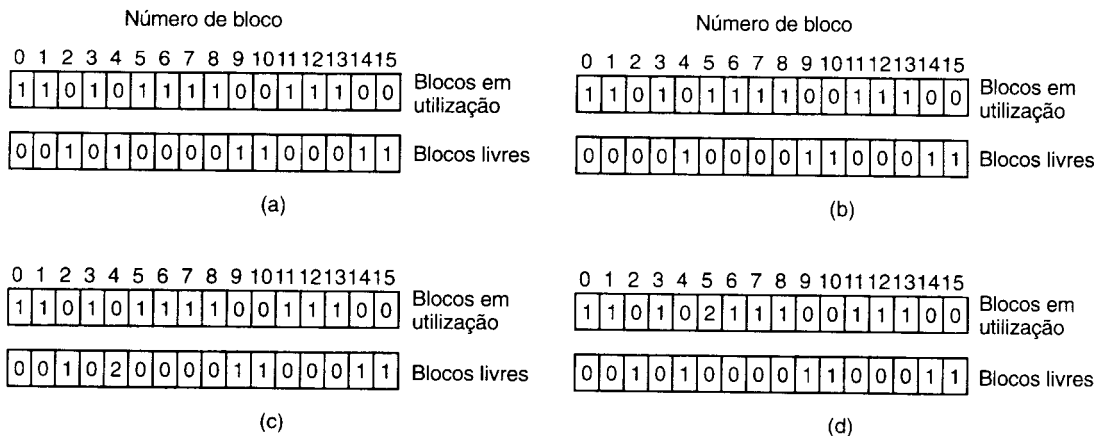



Figura 5-18 Estados do sistema de arquivos. (a) Consistente. (b) Bloco ausente. (c) Bloco duplicado na lista livre. (d) Bloco de dados duplicado.

A ação apropriada para o verificador de sistema de arquivos tomar é alocar um bloco livre, copiar o conteúdo do bloco 5 nele e inserir a cópia em um dos arquivos. Assim, o conteúdo das informações dos arquivos permanece inalterado (embora quase seguramente embaralhado), mas pelo menos a estrutura do sistema de arquivos é tornada consistente. O erro deve ser informado, permitindo que o usuário inspecione o dano.

Além de verificar se cada bloco está adequadamente contabilizado, o verificador do sistema de arquivos também verifica o sistema de diretórios. Ele também utiliza uma tabela de contadores, mas esses são por arquivo, em vez de por bloco. Ele inicia no diretório-raiz e recursivamente desce a árvore, inspecionando cada diretório no sistema de arquivos. Para cada arquivo em cada diretório, ele incrementa o contador para o nó-i desse arquivo (veja a Figura 5-13 para o arranjo de uma entrada de diretório).

Quando tudo está feito, ele tem uma lista, indexada por número de nó-i, dizendo quantos diretórios apontam para esse nó-i. Então, ele compara esses números com a conta de vínculos armazenada nos próprios nós-i. Em um sistema de arquivos consistente, as duas contagens coincidirão. Entretanto, dois tipos de erro podem ocorrer: a contagem de vínculos no nó-i pode ser muito alta ou muito baixa.

Se a contagem de vínculos é mais alta do que o número de entradas de diretório, então, mesmo se todos os arquivos forem removidos dos diretórios, a contagem ainda será não-zero e o nó-i não será removido. Esse erro não é sério, mas desperdiça espaço no disco com arquivos que não estão em nenhum diretório. Isso deve ser corrigido, configurando-se a contagem de vínculos no nó-i com o valor correto.

O outro erro é potencialmente catastrófico. Se duas entradas de diretório estão vinculadas a um arquivo, mas o nó-i diz que há somente uma, quando qualquer uma das entradas de diretório for removida, a contagem do nó-i irá zerar. Quando a contagem do nó-i chega a zero, o sistema de arquivos marca-o como não-utilizado e libera todos os seus blocos. Essa ação fará com que um dos diretórios agora aponte para um nó-i não-utilizado, cujos blocos seguintes podem ser atribuídos a outros arquivos. Novamente, a solução é simplesmente forçar a contagem de vínculos no nó-i para o número real de entradas de diretório.

Essas duas operações, verificar blocos e verificar diretórios, freqüentemente são integradas por razões de eficiência (i. e., somente uma passagem sobre os nós-i é exigida). Outra verificações heurísticas são também possíveis. Por exemplo, os diretórios têm um formato definido, com números de nó-i e nomes ASCII. Se um número de nó-i for maior que o número de nós-i no disco, o diretório foi danificado.

Além disso, cada nó-i tem um modo, alguns dos quais são válidos, mas estranhos, como 0007, que não concede ao proprietário e a seu grupo absolutamente nenhum acesso, mas permite que estranhos leiam, gravem e executem o arquivo. Talvez seja útil pelo menos informar sobre arquivos que dão a estranhos mais direitos do que ao proprietário. Os diretórios com mais que, digamos, 1.000 entradas também são suspeitos. Arquivos localizados em diretórios de usuário, mas que são possuídos pelo superusuário e tem o bit de SETUID ativado, são problemas de segurança potenciais. Com um pequeno esforço, pode-se montar uma lista relativamente longa de situações legais, mas peculiares, que talvez mereçam um informe.

Os parágrafos anteriores discutiram o problema de proteger o usuário contra quedas. Alguns sistemas de arquivos também preocupam-se em proteger o usuário contra si próprio. Se o usuário pretender digitar

```
rm *.o
```

para remover todos os arquivos que terminam com *.o* (arquivos-objeto gerados pelo compilador), mas acidentalmente digitar

```
rm * .o
```

(note o espaço depois do asterisco), *rm* removerá todos os arquivos no diretório atual e, então, irá queixar-se que não pôde localizar *.o*. No MS-DOS e em alguns outros sistemas, quando um arquivo é removido, tudo o que acontece é que um bit é configurado no diretório ou no nó-i, marcando o arquivo como removido. Nenhum bloco de disco é retornado à lista de livres até que eles realmente sejam necessários. Assim, se o usuário descobre o erro imediatamente, é possível executar um programa especial utilitário que "desexclui" (i. e., restaura) os arquivos removidos. No WINDOWS 95, arquivos que são removidos são colocados em um diretório especial *recycled* (*lixeira*), do qual eles podem ser recuperados mais tarde se necessário. Naturalmente, nenhum armazenamento é reivindicado até que eles sejam realmente excluídos desse diretório.

5.3.5 Desempenho do Sistema de Arquivos

O acesso a disco é muito mais lento que o acesso à memória. A leitura de uma palavra da memória geralmente leva dezenas de nanossegundos. A leitura de um bloco de um disco rígido pode levar 50 microssegundos, um fator quatro vezes mais lento por palavra de 32 bits, mas a isso deve ser adicionado 10 a 20 milissegundos para buscar a trilha e, então, esperar o setor desejado chegar sob o cabeçote de leitura. Se apenas uma única palavra for necessária, o acesso à memória é da ordem de 100.000 vezes mais rápido que o acesso a disco. Como um resultado dessa diferença em tempo de acesso, muitos sistemas de arquivos foram projetados para reduzir o número de acessos a disco necessários.

A técnica mais comum utilizada para reduzir acessos a disco é o ***cache* de bloco** ou o ***cache* de buffer.** (*Cache* é pronunciado como "kàsh", e deriva do francês *cacher*, que significa *ocultar*.) Nesse contexto, um *cache* é uma coleção de blocos que logicamente pertencem ao disco, mas que estão sendo mantidos na memória por razões de desempenho.

Vários algoritmos podem ser utilizados para gerenciar o *cache*, mas um comum é verificar todas as solicitações de leitura para ver se o bloco necessário está no *cache*. Se estiver, a solicitação de leitura pode ser satisfeita sem acesso de disco. Se o bloco não estiver no *cache*, é primeiro lido no *cache* e, então, copiado para qualquer lugar que seja necessário. Solicitações subseqüentes para o mesmo bloco podem ser satisfeitas a partir do *cache*.

Quando um bloco precisa ser carregado em um *cache* cheio, algum bloco precisa ser removido e regravado para o disco se foi modificado desde que foi trazido. Essa situação é muito parecida com a paginação, e todos os algoritmos normais de paginação descritos no Capítulo 4, como FIFO, segunda chance, e LRU, são aplicáveis. Uma diferença agradável entre paginação e *cache* é que referências de *cache* são relativamente raras, de modo que é razoável manter todos os blocos na ordem exata de LRU com listas encadeadas.

Infelizmente, há uma cilada. Agora que temos uma situação em que LRU exato é possível, este revela-se indesejável. O problema tem a ver com as quedas e a consistência do sistema de arquivos discutidas na seção anterior. Se um bloco crítico, como um bloco de nó-i, é lido no *cache* e modificado, mas não-regravado no disco, uma queda deixará o sistema de arquivos em um estado inconsistente. Se o bloco de nó-i tiver sido colocado no fim da cadeia de LRU, pode demorar um pouco para alcançar a frente e ser regravado no disco.

Além disso, alguns blocos, como os indiretos duplos, raramente são referenciados duas vezes dentro de um intervalo curto. Essas considerações conduzem a um esquema modificado de LRU, levando em conta dois fatores:

1. É possível que o bloco seja necessário novamente em breve?
2. O bloco é essencial para a consistência do sistema de arquivos?

Para ambas as perguntas, os blocos podem ser divididos em categorias como blocos de nó-i blocos indiretos, como blocos de diretório, como blocos cheios de dados e blocos parcialmente cheios de dados. Os blocos que provavelmente não serão necessários novamente irão na frente, em vez de no fim da lista de LRU, portanto, seus buffers serão reutilizados rapidamente. Os blocos que talvez sejam necessários novamente logo, como um bloco parcialmente cheio que está sendo gravado, entram no fim da lista, para que permaneçam à mão por bastante tempo.

A segunda pergunta é independente da primeira. Se o bloco é essencial para a consistência do sistema de arquivos (basicamente, tudo exceto blocos de dados) e foi modificado, ele deverá ser gravado em disco imediatamente, independentemente da extremidade da lista de LRU em que ele foi colocado. Gravando blocos críticos com rapidez, reduzimos significativamente a probabilidade de que uma queda destruirá o sistema de arquivos.

Mesmo com essa medida para manter intacta a integridade do sistema de arquivos, é indesejável manter blocos de dados no *cache* por muito tempo antes de gravá-los. Considere o compromisso de alguém que esteja utilizando um computador pessoal para escrever um livro. Mesmo se nosso escritor periodicamente instruísse o editor para gravar no disco o arquivo que está sendo editado, há uma boa chance de que tudo ainda estará no *cache* e nada no disco. Se o sistema cai, a estrutura do sistema de arquivos não será corrompida, mas um dia inteiro de trabalho será perdido.

Essa situação não precisa acontecer com muita freqüência para termos um usuário insatisfeito. Para lidar com ela, os sistemas exigem duas abordagens. A maneira do UNIX é ter uma chamada de sistema, SYNC, que força todos os blocos modificados a ir para o disco imediatamente. Quando o sistema é iniciado, um programa, normalmente chamado *update*, é iniciado em segundo plano para permanecer em um laço interminável que emite chamadas SYNC, dormindo por 30 s entre as chamadas. Como resultado, não

mais que 30 segundos de trabalho é perdido devido a uma queda.

A maneira do MS-DOS é gravar em disco cada bloco modificado logo que ele foi gravado. Os *caches* em que todos os blocos modificados são gravados de volta para o disco imediatamente são chamados ***caches* de gravação intermediária**. Eles exigem muito mais E/S de disco do que os outros tipos de *cache*. A diferença entre essas duas abordagens pode ser vista quando um programa grava um bloco de 1K cheio, um caractere por vez. O UNIX reunirá todos os caracteres no *cache* e gravará o bloco no disco uma vez a cada 30 segundos ou sempre que o bloco for removido do *cache*. O MS-DOS fará um acesso de disco a cada caractere gravado. Naturalmente, a maioria dos programas faz *bufferização* interna, portanto, eles normalmente não gravam um caractere, mas uma linha ou uma unidade maior a cada chamada de sistema WRITE.

Uma conseqüência dessa diferença nas estratégias de fazer *cache*, é que a simples remoção de um disco (disquete) de um sistema UNIX sem fazer um SYNC quase sempre resultará em dados perdidos e, freqüentemente, em um sistema de arquivos corrompido também. Com o MS-DOS, nenhum problema surge. Essas estratégias diferentes foram escolhidas porque o UNIX foi desenvolvido em um ambiente em que todos os discos eram discos rígidos e não-removíveis, enquanto o MS-DOS começou com disquetes. À medida que os discos rígidos tornam-se a norma, mesmo em microcomputadores pequenos, a abordagem do UNIX, com sua melhor eficiência, será definitivamente a maneira recomendada e adotada.

Fazer *cache* não é a única maneira de aumentar o desempenho de um sistema de arquivos. Outra técnica importante é reduzir o deslocamento do braço de disco, colocando os blocos que podem ser acessados em seqüência próximos um do outro, preferivelmente no mesmo cilindro. Quando um arquivo de saída é gravado, o sistema de arquivos precisa alocar os blocos um por vez, conforme eles sejam necessários. Se os blocos livres são registrados em um mapa de bits e o mapa de bits inteiro está na memória principal, é muito fácil escolher um bloco livre o mais próximo possível do bloco anterior. Com uma lista de livres, parte da qual está em disco, é muito mais difícil alocar blocos próximos ou juntos.

Entretanto, mesmo com uma lista de livres, alguma alocação de bloco contíguos pode ser feita. O truque é monitorar o armazenamento de disco não em blocos, mas em grupos de blocos consecutivos. Se uma trilha consiste em 64 setores de 512 bytes, o sistema poderia utilizar blocos de 1K (2 setores), mas alocar armazenamento de disco em unidades de 2 blocos (4 setores). Isso não é o mesmo que ter um bloco de disco de 2K, uma vez que o *cache* ainda utilizaria blocos de 1K, e as transferências de disco ainda seriam de 1K, mas a leitura seqüencial de um arquivo em um sistema que, por outro lado, estaria desocupado, reduziria o número de buscas por um fator de dois, melhorando consideravelmente o desempenho.

Uma variação sobre o mesmo tema é levar em conta o posicionamento rotacional. Ao alocar blocos, o sistema tenta colocar blocos consecutivos em um arquivo no mesmo cilindro, mas de maneira intercalada para alcançar tangência máxima. Portanto, se um disco tiver um tempo de rotação de 16,67ms e levar aproximadamente 4ms para um processo de usuário solicitar e obter um bloco de disco, cada bloco deve ser colocado a pelo menos 1/4 da distância até seu predecessor.

Outro gargalo de desempenho em sistemas utilizando nós-i ou qualquer coisa equivalente a nós-i, é que mesmo a leitura de um arquivo curto requer dois acessos a disco: um para o nó-i e um para o bloco. A colocação normal de nó-i é mostrada na Figura 5-19(a). Aqui todos os nós-i estão próximos do começo do disco, então, a distância média entre um nó-i e seus blocos será a metade do número de cilindros, exigindo buscas longas.

Uma maneira fácil de melhorar o desempenho é colocar os nós-i no meio do disco, em vez de no início, reduzindo assim a busca média entre o nó-i e o primeiro bloco por um fator de dois. Outra idéia, mostrada na Figura5-19(b), é dividir o disco em grupos de cilindros, cada um com seus próprios nós-i, blocos e lista de livres (McKusick *et al.*, 1984). Ao criar um novo arquivo, qualquer nó-i pode ser escolhido, mas é feita uma tentativa de localizar um bloco no mesmo grupo de cilindros que o nó-i. Se nenhum estiver disponível, então, um bloco perto do grupo de cilindros é utilizado.

5.3.6 Sistemas de Arquivos Estruturados em *Log*

As mudanças na tecnologia estão exercendo uma pressão nos sistemas de arquivos atuais. Em particular, as CPUs ficam cada vez mais rápidas, os discos estão tornando-se cada vez maiores e mais baratos (mas não muito mais rápidos) e as memórias estão crescendo exponencialmente em tamanho. O parâmetro que não está melhorando aos saltos é o tempo de busca do disco. A combinação desses fatores significa que um gargalo de desempenho está surgindo em muitos sistemas de arquivos. Pesquisas feitas em Berkeley tentaram aliviar esse problema projetando um tipo completamente novo de sistema de arquivos, **LFS** (***Log-Structured File System***/Sistemas de Arquivos Estruturados em *Log*). Nesta seção, descreveremos resumidamente como o LFS funciona. Para um tratamento mais completo, veja (Rosenblum e Ousterhout, 1991).

A idéia que guiou o projeto do LFS é que, à medida que as CPUs ficam mais rápidas, e as memórias RAM ficam maiores, os *caches* de disco tornam-se cada vez mais rápidos. Como conseqüência, agora é possível satisfazer uma fração muito substancial de toda solicitação de leitura diretamente do *cache* do sistema de arquivos, sem necessidade de acessos a disco. Deduz-se dessa observação que, no futuro, a maioria dos acessos de disco será de gravação, portanto, o mecanismo de leitura antecipada (*read-ahead*),

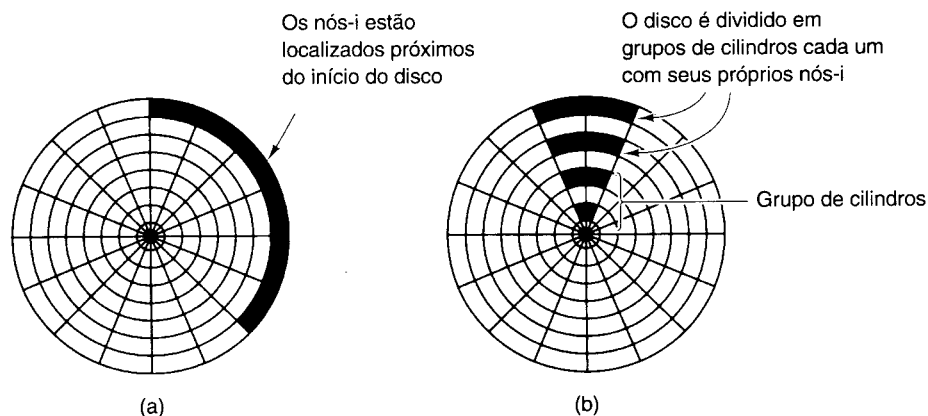



Figura 5-19 (a) Nós-i colocados no início do disco. (b) Disco dividido em grupos de cilindros, cada um com seus próprios blocos e nós-i.

utilizado em alguns sistemas de arquivos para buscar blocos antes de eles serem necessários, não mais oferece muito ganho de desempenho.

Para piorar as coisas, na maioria dos sistemas de arquivos, as gravações são feitas em porções muito pequenas. Gravações pequenas são altamente ineficientes, uma vez que uma gravação em disco de 50 microssegundos geralmente é precedida por uma busca de 10ms, e um retardo rotacional de 6ms. Com esses parâmetros, a eficiência de disco cai para uma fração de 1%.

Para ver de onde provêm todas as gravações pequenas, considere a criação de um novo arquivo em um sistema UNIX. Para gravar esse arquivo, o nó-i para o diretório, o bloco de diretório, o nó-i para o arquivo e o próprio arquivo devem ser todos gravados. Embora essas gravações possam ser retardadas, fazer isso expõe o sistema de arquivos a problemas sérios de consistência se uma queda ocorrer antes das gravações terem sido feitas. Por essa razão, as gravações de nó-i geralmente são feitas imediatamente.

A partir desse raciocínio, os projetistas do LFS decidiram reimplementar o sistema de arquivos do UNIX de maneira a alcançar toda a largura de banda do disco, mesmo perante uma carga de trabalho que, em grande parte, consiste de pequenas gravações aleatórias. A idéia básica é estruturar o disco inteiro como um *log*. Periodicamente e quando houver uma necessidade especial, todas as gravações pendentes sendo ***bufferizadas*** na memória são colecionadas em um único segmento e gravadas no disco como em um único segmento contínuo no fim do *log*. Um segmento único pode assim conter nós-i, blocos de diretório e blocos de dados, todos misturados. No início de cada segmento, está um resumo do segmento, informando o que pode ser encontrado no segmento. Se o segmento médio pode ser levado a ser de aproximadamente 1MB, quase toda a largura de banda do disco pode ser utilizada.

Nesse projeto, nós-i ainda existem e têm a mesma estrutura que no UNIX, mas agora estão dispersos por todo o *log*, em vez de estar em uma posição fixa no disco. Contudo, quando um nó-i é localizado, localizar os blocos é feito na maneira usual. Naturalmente, localizar um nó-i é agora muito mais difícil, uma vez que seu endereço simplesmente não pode ser calculado a partir de seu número i, como no UNIX. Para tornar possível localizar nós-i, um mapa de nós-i, indexado pelo número *i*, é mantido. A entrada *i* desse mapa aponta para o nó-i *i* no disco. O mapa é mantido em disco, mas também sofre *cache*, então, partes mais intensamente utilizadas estarão na memória na maior parte do tempo.

Para resumir o que dissemos até agora, todas as gravações são armazenadas na memória e periodicamente todas as gravações armazenadas são gravadas no disco em um único segmento, no fim do *log*. A abertura de um arquivo agora consiste em utilizar o mapa para localizar o nó-i do arquivo. Uma vez que o nó-i foi localizado, os endereços dos blocos podem ser localizados a partir dele. Todos os blocos em si estarão em segmentos, em algum lugar no *log*.

Se os discos fossem infinitamente grandes, a descrição acima encerraria a história. Contudo, os discos reais são limitados, então, por fim, o *log* acaba ocupando o disco inteiro, momento em que nenhum novo segmento pode ser gravado no *log*. Felizmente, muitos segmentos existentes podem ter blocos que não são mais necessários, por exemplo, se um arquivo é sobrescrito, seu nó-i agora apontará para os novos blocos, mas os antigos ainda estarão ocupando espaço nos segmentos anteriormente gravados.

Para lidar com esses dois problemas, o LFS tem um *thread* **limpador** que gasta seu tempo varrendo o *log* circularmente para compactá-lo. Ele começa lendo o resumo do primeiro segmento no *log* para ver quais nós-i e quais arquivos estão aí. Então, ele verifica o mapa atual de nós-i para ver se os nós-i são ainda atuais e se os blocos de arquivo ainda estão em uso. Se não, as informações são descartadas. Os nós-i e os blocos que ainda estão em uso são trazidos para a memória para serem gravados em disco no

próximo segmento. O segmento original, então, é marcado como livre, portanto, o *log* pode utilizá-lo para novos dados. Dessa maneira, o limpador move-se ao longo do *log*, removendo segmentos antigos do final e colocando quaisquer dados em uso na memória para regravação no próximo segmento. Conseqüentemente, o disco é um grande buffer circular, com o *thread* gravador adicionando novos segmentos na frente, e os *thread* limpadores removendo os antigos do final.

A manutenção aqui não é trivial, uma vez que quando um bloco de arquivo é gravado de volta em um novo segmento, o nó-i do arquivo (em algum lugar no *log*) deve ser localizado, atualizado e colocado na memória para ser gravado no próximo segmento. O mapa de nó-i, então, deve ser atualizado para apontar para a nova cópia. Contudo, é possível fazer a administração, e os resultados de desempenho mostram que toda essa complexidade é vantajosa. As medidas dadas nos papers citados anteriormente mostram que o desempenho do LFS superam o do UNIX em uma ordem de magnitude em pequenas gravações, enquanto têm um desempenho que é tão bom ou melhor que o do UNIX para leituras e para gravações grandes.

5.4 SEGURANÇA

Os sistemas de arquivos com freqüência contêm informações extremamente valiosas para seus usuários. Proteger essas informações contra o uso não-autorizado é, portanto, uma questão importante para todos os sistemas de arquivos. Nas seções a seguir veremos diversas questões relacionadas com segurança e com proteção. Essas questões aplicam-se igualmente à maioria dos sistemas de compartilhamento de tempo, bem como a redes de computadores pessoais conectadas a servidores compartilhados via redes locais.

5.4.1 Ambiente de Segurança

Os termos "segurança" e "proteção" são freqüentemente utilizados de maneira intercambiável. Contudo, com freqüência, é útil fazer uma distinção entre os problemas gerais envolvidos em assegurar-se de que os arquivos não sejam lidos nem modificados por pessoas não-autorizadas, o que inclui questões políticas, legais, administrativas e técnicas, de um lado, e os mecanismos específicos do sistema operacional utilizados para oferecer segurança, do outro. Para evitar confusão, utilizaremos o termo **segurança** para referirmo-nos ao problema total, e o termo **mecanismos de proteção** para referirmo-nos aos mecanismos específicos do sistema operacional utilizados para salvaguardar as informações no computador. A linha divisória entre eles, entretanto, não é bem-definida. Primeiro veremos segurança; mais adiante no capítulo veremos proteção.

A segurança tem muitos aspectos. Dois dos mais importantes são a perda de dados e os intrusos. Algumas causas comuns de perda de dados são:

1. Ações divinas: incêndios, inundações, terremotos, guerras, revoltas ou ratos que roem fitas ou disquetes.
2. Erros de hardware ou de software: malfuncionamento da CPU, discos ou fitas ilegíveis, erros de telecomunicação, *bugs* de programa.
3. Erros humanos: entrada incorreta de dados, montagem incorreta de fita ou de disco, execução errada de programa, perda de disco ou de fita ou algum outro engano.

A maioria desses pode ser tratada mantendo-se backups adequados, preferivelmente longe dos dados originais.

Um problema mais interessante é o que fazer com intrusos. Esses se classificam em dois tipos. Intrusos passivos somente querem ler arquivos que eles não são autorizados a ler. Intrusos ativos são mais maliciosos; eles querem fazer alterações não-autorizadas nos dados. Ao projetar um sistema para ser seguro contra intrusos, é importante ter em mente o tipo de intruso contra o qual se está tentando criar proteção. Algumas categorias comuns são:

1. Bisbilhotice casual por usuários não-técnicos. Muitas pessoas têm terminais para sistemas de compartilhamento de tempo ou para computadores pessoais em rede em suas mesas, e a natureza humana sendo a que é, algum deles lerão correio eletrônico e outros arquivos de outras pessoas se nenhuma barreira for colocada no caminho. A maioria dos sistemas UNIX, por exemplo, tem como padrão que todos os arquivos são publicamente legíveis.
2. Espionagem por pessoas de dentro. Alunos, programadores de sistema, operadores e outro pessoal técnico, com freqüência, consideram ser um desafio pessoal quebrar a segurança do sistema local de computador. Eles seguidamente são bastante habilidosos e estão disposto a dedicar uma quantidade significativa de tempo nesse esforço.
3. Tentativa determinada de fazer dinheiro. Alguns programadores de instituições bancárias tentaram quebrar um sistema de depósitos para roubar um banco. Os esquemas variavam desde a alteração do software para truncar em vez de arredondar a taxa de juros, mantendo as frações de um centavo para eles próprios, até o furto de contas não-utilizadas há anos, e chantagem ("Paguem-me ou destruirei todos os registros do banco".)
4. Espionagem comercial ou militar. Espionagem refere-se a uma tentativa mais séria e bem-financiada por um concorrente ou por um país estrangeiro para roubar programas, segredos de negócio, patentes, tecnologia, projetos de circuitos, planos de *marketing*, etc. Freqüentemente essa tentativa envolverá grampeamento ou mesmo montar antenas dirigidas para o computador a fim de captar sua radiação electromagnética.

Deve estar claro que tentar impedir um governo estrangeiro hostil de roubar segredos militares é uma questão bem diferente de tentar impedir alunos de inserir uma "mensagem do dia engraçada" no sistema. A quantidade de esforço aplicada em segurança e proteção depende muito do inimigo considerado.

Outro aspecto do problema da segurança é a **privacidade**: proteger indivíduos do abuso das informações sobre eles. Isso rapidamente leva a muitas questões morais e jurídicas. O governo deve compilar dossiês sobre todo mundo para capturar fraudadores de *x*, onde *x* é "previdência social" ou "imposto de renda", dependendo da sua política? A polícia deve ser capaz de pesquisar qualquer coisa sobre qualquer pessoa para combater o crime organizado? Os empregadores e as companhias de seguro têm direitos? O que acontece quando esses direitos entram em conflito com os direitos do indivíduo? Todas essas questões são extremamente importantes, mas estão além do âmbito deste livro.

5.4.2 Falhas Famosas de Segurança

Assim como a indústria dos transportes tem seu *Titanic* e o seu *Hindenburg*, os peritos de segurança de computador têm algumas coisas das quais jamais se esquecerão. Nesta seção, veremos alguns problemas interessantes de segurança que ocorreram em três sistemas operacionais diferentes: UNIX, TENEX e OS/360.

O utilitário *lpr* do UNIX, que imprime um arquivo na impressora de linha, tem uma opção para remover o arquivo depois que ele foi impresso. Em versões primitivas do UNIX, era possível qualquer pessoa utilizar *lpr* para imprimir e, então, fazer o sistema remover o arquivo de senhas.

Outra maneira de invadir o UNIX era vincular um arquivo chamado *core* no diretório de trabalho ao arquivo de senha. O intruso, então, forçava um *dump* de núcleo de um programa SETUID, que o sistema gravava no arquivo *core*, isto é, sobre o arquivo de senhas. Dessa maneira, um usuário poderia substituir o arquivo de senhas por outro contendo algumas *strings* da sua própria escolha (p. ex., argumentos de comando).

Ainda, outra falha sutil no UNIX envolvia o comando

```
mkdir foo
```

mkdir, que era um programa SETUID possuído pela raiz, primeiro criava o nó-i para o diretório *foo* com a chamada de sistema MKNOD e, então, alterava o proprietário de *foo* de seu *uid* efetivo (i. e., raiz) para seu *uid* real (o *uid* do usuário). Quando o sistema era lento, às vezes, era possível ao usuário rapidamente remover o nó-i do diretório e fazer um *link* para o arquivo de senha sob o nome de *foo* depois do MKNOD, mas antes do CHOWN. Quando *mkdir* fazia o CHOWN, o usuário tornava-se o proprietário do arquivo de senha. Colocando os comandos necessários em um *script* de *shell*, isso podia ser feito repetidamente até o truque funcionar.

O sistema operacional TENEX costumava ser muito popular nos computadores DEC-10. Ele não é mais utilizado, mas sobreviverá eternamente nos anais de segurança de computador devido ao seguinte erro de projeto. O TENEX suportava paginação. Para permitir que os usuários monitorassem o comportamento de seus programas, era possível instruir o sistema a chamar uma função de usuário a cada falha de página.

O TENEX também utilizava senhas para proteger os arquivos. Para acessar um arquivo, um programa precisava apresentar a senha adequada. O sistema operacional verificava as senhas um caractere por vez, parando logo que via que a senha estava errada. Para penetrar no TENEX, um intruso cuidadosamente posicionaria uma senha como mostrado na Figura 5-20(a), com o primeiro caractere no fim de uma página e o restante no início da próxima página.

O próximo passo era certificar-se de que a segunda página não estivesse na memória, por exemplo, referencian-

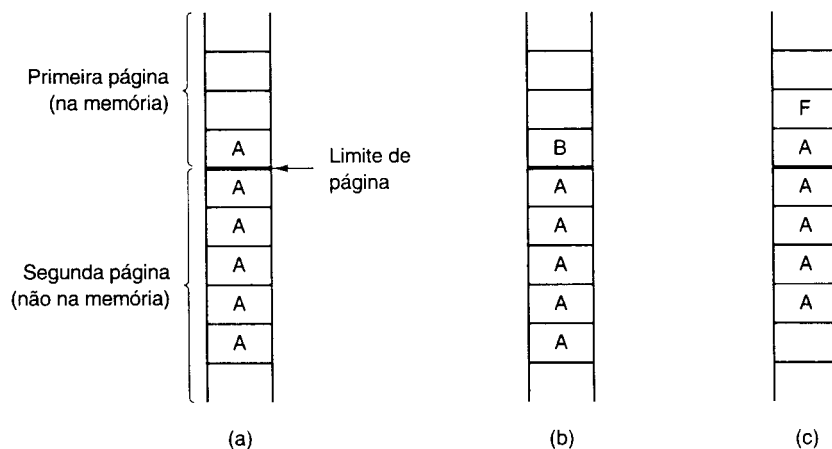



Figura 5-20 O problema de senha do TENEX.

do tantas outras páginas que a segunda página seria seguramente expulsa para dar lugar a elas. Agora o programa tentava abrir o arquivo da vítima, utilizando a senha cuidadosamente alinhada. Se o primeiro caractere da senha real fosse qualquer coisa exceto *A*, o sistema pararia para verificar o primeiro caractere e retornaria um informe de ILLEGAL PASSWORD. Se, entretanto, a senha real começasse com *A*, o sistema continuava a leitura e obtinha uma falha de página, sobre a qual o intruso era informado.

Se a senha não começasse com *A*, o intruso alterava a senha para a da Figura 5-20(b) e repetia o processo inteiro para ver se começava com *B*. No máximo, 128 tentativas seriam necessárias para percorrer todo o conjunto de caracteres ASCII e assim determinar o primeiro caractere.

Supondo que o primeiro caractere fosse um *F*. O arranjo de memória da Figura 5-20(c) permitia que o intruso testasse cadeias de caracteres na forma *FA*, *FB* e assim por diante. Utilizando essa abordagem que levava no máximo $128n$ tentativas para adivinhar uma senha de n caracteres ASCII, em vez de 128^n.

Nossa última falha diz respeito ao OS/360. A descrição que se segue é ligeiramente simplificada, mas conserva a essência da falha. Esse sistema podia iniciar uma leitura de fita e, então, continuar a computação enquanto a unidade de fita estava transferindo dados para o espaço do usuário. O truque aqui é cuidadosamente iniciar uma leitura de fita e, então, fazer uma chamada de sistema que solicitava uma estrutura de dados do usuário, por exemplo, um arquivo para ler e sua senha.

O sistema operacional primeiro verificava se a senha era de fato a correta para o arquivo dado. Então, ele voltava e lia o nome do arquivo novamente para o acesso real (ele podia salvar o nome internamente, mas não fazia isso). Infelizmente, logo antes de o sistema buscar o nome do arquivo pela segunda vez, o nome do arquivo era sobrescrito pela unidade de fita. O sistema, então, lia o novo arquivo, para o qual nenhuma senha fora apresentada. Para conseguir a sincronização correta era necessária alguma prática, mas isso não era tão difícil. Além disso, se há uma coisa em que computadores são bons, é executar a mesma operação repetidamente *ad nauseam*.

Além desses exemplos muitos outros problemas de segurança e de ataques surgiram com os anos. Um que apareceu em muitos contextos é o **cavalo de Tróia**, no qual um programa aparentemente inocente que é amplamente distribuído também executa alguma função indesejável e inesperada, como roubar dados e enviá-los por correio eletrônico para algum *site* distante onde podem ser reunidos mais tarde.

Outro problema de segurança nesses tempos de insegurança de trabalho é o da **bomba lógica**. Esse dispositivo é um pequeno código escrito por um dos programadores de uma empresa (no momento sendo empregado) e secretamente inserido no sistema operacional de produção. Contanto que o programador alimente-o com sua senha diariamente, ele não faz nada. Entretanto, se o programador repentinamente é despedido e fisicamente removido das premissas sem aviso, no dia seguinte em que a bomba lógica não recebe sua senha, ela dispara.

O disparo talvez envolva limpar o disco, apagar arquivos aleatoriamente, fazer cuidadosas alterações difíceis de detectar em programas-chave ou criptografar arquivos essenciais. Neste último caso, a empresa precisará fazer uma difícil escolha entre chamar a polícia (que pode ou não resultar em uma condenação muitos meses mais tarde) ou ceder a essa chantagem e recontratar o ex-programador como um "consultor" por uma soma astronômica para corrigir o problema (e esperar que ele não plante novas bombas lógicas enquanto faz isso).

Provavelmente a maior violação da segurança de computadores de todos os tempos começou na noite de 2 de novembro de 1988 quando um aluno graduado em Cornell, Robert Tappan Morris, lançou um programa-verme na Internet que acabou derrubando milhares de máquinas por todo o mundo.

O verme consistia em dois programas, o comando de partida (*bootstrap*) e o verme em si. O programa de partida era constituído por 99 linhas em C de chamadas *l1.c*. Ele era compilado e executado no sistema sob ataque. Uma vez em execução, ele se conectava à máquina de que veio, carregava o verme principal e executava-o. Após passar por algumas etapas complicadas para ocultar sua existência, o verme, então, pesquisava as tabelas de roteamento do seu novo *host* para ver com que máquinas esse *host* estava conectado e tentava espalhar o comando de partida para essas máquinas.

Uma vez estabelecido em uma máquina, o verme tentava quebrar as senhas de usuário. Morris não precisou pesquisar muito para descobrir como fazer isso. Tudo que ele precisou fazer foi pedir a seu pai, um perito de segurança na Agência Nacional de Segurança, o supersecreto órgão do governo norte-americano encarregado de quebrar códigos de segurança, uma reimpressão de um paper clássico sobre o assunto que Sr. Morris e Ken Thompson tinham escrito uma década antes no Bell Labs (Morris e Thompson, 1979). Cada senha quebrada permitia que o verme se conectasse a quaisquer máquinas em que o proprietário da senha tivesse contas.

Morris foi capturado quando um de seus amigos falou com o repórter de informática do *New York Times*, John Markoff, e tentou convencer Markoff de que o incidente foi um acidente, o verme era inofensivo e o autor lamentava muito tudo aquilo. No dia seguinte, a história ganhou as manchetes dos jornais, tirando a atenção até mesmo da eleição presidencial três dias depois. Morris foi julgado e condenado na corte suprema. Ele foi sentenciado a uma multa de 10 mil dólares, três anos de condicional e 400 horas de serviços comunitários. Seus custos advocatícios provavelmente excederam 150 mil dólares.

Essa sentença gerou muita controvérsia. Muitos na comunidade da informática acreditavam que ele era um brilhante aluno da graduação cuja inofensiva brincadeira tinha saído do seu controle. Nada no verme de Morris sugeria que ele estava tentando roubar ou danificar qualquer

coisa. Outros o consideravam um criminoso sério que deveria ir para a cadeia.

Um efeito permanente desse incidente foi o estabelecimento do **CERT** (***Computer Emergency Response Team*** — equipe de resposta à emergências relacionadas a computadores), que oferece uma central de informações sobre tentativas de invasão, e um grupo de peritos para analisar problemas de segurança e desenvolver soluções para tais problemas. Embora essa ação certamente tenha dado um passo a frente, ela também deu um passo para trás. O CERT reúne defeitos de sistemas que podem ser atacados e informações sobre como corrigi-los. Em função da necessidade, ele faz circular amplamente essas informações para milhares de administradores de sistema na Internet, o que significa que também os mal-intencionados podem ser capazes de obtê-las e explorar as brechas nas horas (ou mesmo dias) antes de o acesso ser fechado.

5.4.3 Ataques de Segurança Genéricos

As falhas descritas acima foram corrigidas, mas o sistema operacional médio ainda tem mais lacunas do que uma peneira. A maneira normal de testar a segurança de um sistema é empregar um grupo de peritos, conhecidos como **equipes de invasão**, para ver se eles podem quebrá-la. Hebbard e colaboradores (1980) tentaram a mesma coisa com alunos de graduação. No curso dos anos, essas equipes de invasão descobriram diversas áreas em que os sistemas podem ser vulneráveis. A seguir, listamos alguns ataques mais comuns que freqüentemente são bem-sucedidos. Ao projetar um sistema, esteja seguro de que pode combater ataques como estes.

1. Solicite páginas de memória, espaço em disco ou fitas e simplesmente os leia. Muitos sistemas não os apagam antes de alocá-los e eles podem estar cheios de informações interessantes, gravadas pelo proprietário anterior.
2. Tente chamadas de sistema ilegais ou chamadas de sistema legais com parâmetros ilegais ou chamadas de sistema legais comuns com parâmetros legais, mas improváveis. Muitos sistemas facilmente podem ser confundidos.
3. Inicie efetuando *logon* e, então, pressione DEL, RUBOUT ou BREAK no meio da seqüência de *login*. Em alguns sistemas, o programa que verifica a senha será eliminado, e o *login* considerado bem-sucedido.
4. Tente modificar as estruturas complexas mantidas pelo sistema operacional no espaço do usuário (se houver alguma). Em alguns sistemas (especialmente em *mainframes*), para abrir um arquivo, o programa constrói uma grande estrutura de dados que contém o nome de arquivo e de muitos outros parâmetros, e passa-os para o sistema. Enquanto o arquivo é lido e gravado, o sistema, às vezes, atualiza a própria estrutura. A alteração desses campos pode devastar a segurança.
5. Experimente ludibriar o usuário, escrevendo um programa que escreva "*login*": na tela e segue adiante. Muitos usuários irão até o terminal e diligentemente informarão seu nome de *login* e sua senha, que o programa cuidadosamente grava para seu maligno mestre.
6. Procure manuais que dizem "não faça *X*". Tente o máximo de variações possíveis de *x*.
7. Convença um programador de sistema a alterar o sistema para burlar certas verificações de segurança vitais para qualquer usuário com seu nome de *login*. Esse ataque é conhecido como **porta de interrupção (ou porta dos fundos)**.
8. Se tudo isso falhar, o invasor pode encontrar a secretária do diretor do centro de computadores e oferecer-lhe um grande suborno. A secretária provavelmente tem acesso fácil a todo tipo de informações maravilhosas e, em geral, é mal paga. Não subestime problemas causados por funcionários.

Esses e outros ataques são discutidos por Linde (1975).

Vírus

Uma categoria especial de ataque é o vírus de computador, que se tornou um problema importante para muitos usuários de computador. Um **vírus** é um fragmento de programa que é unido a um programa legítimo com a intenção de infectar outros programas. Difere de um verme somente no ponto em que um vírus vale-se de um programa existente, enquanto um verme é um programa completo em si. Vírus e vermes tentam espalhar-se e ambos fazem estragos graves.

Um vírus típico trabalha da seguinte maneira. A pessoa que escreve o vírus primeiro produz um novo programa útil, freqüentemente um jogo para MS-DOS. Esse programa contém o código do vírus oculto dentro dele. O jogo é, então, carregado para um BBS público ou oferecido de graça ou por um preço modesto em disquete. O programa, então, é divulgado, e as pessoas começam a carregá-lo e a utilizá-lo. Construir um vírus não é tão fácil, portanto, as pessoas que fazem isso invariavelmente são bastante inteligentes, e a qualidade do jogo ou outro programa freqüentemente é excelente.

Quando o programa é iniciado, ele imediatamente começa a examinar todos os programas binários no disco rígido para ver se eles já estão infectados. Quando um programa não-infectado é localizado, ele é infectado anexando-se código do vírus ao final do arquivo e substituindo a primeira instrução por um salto para o vírus. Quando o código do vírus termina de executar, ele executa a instrução que anteriormente era a primeira e, então, salta para a segunda instrução. Dessa maneira, toda vez que um pro-

grama infectado executa, ele tenta infectar mais programas.

Além de simplesmente infectar outros programas, um vírus pode fazer outras coisas, como apagar, modificar ou criptografar arquivos. Um vírus chega ao desplante de exibir uma carta de extorsão na tela, dizendo para o usuário enviar 500 dólares em dinheiro para uma caixa de correio no Panamá ou que se conforme com a perda permanente de seus dados e com o estrago do hardware.

Um vírus também pode infectar o setor de inicialização do disco rígido, tornando impossível inicializar o computador. Um vírus assim pode pedir uma senha, que o escritor do vírus pode fornecer em troca de algumas notas pequenas não-marcadas.

Os problemas de vírus são mais fáceis de prevenir do que remediar. O curso seguro é comprar somente software na caixa original, em lojas de confiança. Carregar software livre de BBSs ou obter cópias pirateadas em disquetes é chamar problemas. Existem pacotes comerciais de antivírus, mas alguns desses funcionam pesquisando apenas vírus conhecidos específicos.

Uma abordagem mais geral é primeiro reformatar o disco rígido completamente, incluindo o setor de inicialização. Em seguida, instalar todo o software confiável e calcular uma soma de verificação para cada arquivo. O algoritmo não importa, contanto que tenha bits suficientes (pelo menos 32). Armazene a lista de pares (arquivo, soma de verificação) em um lugar seguro, seja *offline* em um disquete seja *online*, mas criptografado. Iniciando nesse ponto, sempre que o sistema inicializar, todas as somas de verificação devem ser recomputadas e deverão ser comparadas com a lista segura de somas de verificação originais. Qualquer arquivo cuja soma de verificação atual difere da original é imediatamente duvidoso. Embora essa abordagem não impeça a infecção, pelo menos permite detectar cedo sua presença.

A infecção pode ser tornada mais difícil se o diretório onde programas binários residem é tornado não-gravável para usuários comuns. Essa técnica torna difícil o vírus modificar outros binários. Embora possa ser utilizado no UNIX, não é aplicável ao MS-DOS porque os diretórios do último não podem ser tornados não-graváveis de modo algum.

5.4.4 Princípios de Projeto para Segurança

Os vírus ocorrem principalmente em sistemas *desktop*. Em sistemas maiores, outros problemas ocorrem e outros métodos são necessários para lidar com eles. Saltzer e Schroeder (1975) identificaram vários princípios gerais que podem ser utilizados como guia para projetar sistemas seguros. Um breve resumo de suas idéias (baseadas em experiências com o MULTICS) é fornecido a seguir.

Primeiro, o projeto de sistema deve ser público. Assumir que o intruso não saberá como o sistema funciona serve somente para iludir os projetistas.

Segundo, o padrão deve ser nenhum acesso. Os erros em que acesso legítimo é recusado serão informados muito mais rapidamente do que erros em que acesso não-autorizado foi permitido.

Terceiro, verificar a autoridade atual. O sistema não deve verificar permissão, determinar que acesso é permitido e, então, esconder longe essas informações para utilização subseqüente. Muitos sistemas verificam permissão quando um arquivo é aberto e não depois. Isso significa que um usuário que abre um arquivo e mantém-no aberto durante semanas, continuará a ter acesso, mesmo se o proprietário tiver alterado a proteção do arquivo há muito tempo.

Quarto, dê a cada processo o menor privilégio possível. Se um editor tem somente a autoridade para acessar o arquivo a ser editado (especificado quando o editor é invocado), editores com cavalos de Tróia não serão capazes de fazer muito estrago. Esse princípio implica um esquema de proteção refinado. Discutiremos tais esquemas mais adiante neste capítulo.

Quinto, o mecanismo de proteção deve ser simples, uniforme e construído nas camadas mais baixas do sistema. Tentar inserir segurança em um sistema existente inseguro é quase impossível. A segurança, como a precisão, não é um recurso suplementar.

Sexto, o esquema escolhido deve ser psicologicamente aceitável. Se usuários acham que proteger seus arquivos dá muito trabalho, eles simplesmente não farão isso. Contudo, eles se queixarão muito se algo der errado. As respostas na forma "o erro foi seu" geralmente não serão bem-recebidas.

5.4.5 Autenticação do Usuário

Muitos esquemas de proteção são baseados na suposição de que o sistema sabe a identidade de cada usuário. O problema de identificar usuários quando eles se conectam é chamado **autenticação de usuário**. A maioria dos métodos de autenticação é baseada em identificar algo que o usuário sabe, algo que o usuário tem ou algo que o usuário é.

Senhas

A forma mais amplamente utilizada de autenticação é solicitar ao usuário que digite uma senha. A proteção por senhas é fácil de entender e fácil de implementar. No UNIX funciona da seguinte maneira. O programa de *login* solicita que o usuário digite seu nome e sua senha. A senha é imediatamente criptografada. O programa de *login*, então, lê o arquivo de senhas, que é uma série de linhas ASCII, uma por usuário, até que localiza a linha que contém nome de *login* do usuário. Se a senha (criptografada) contida nessa linha coincidir com a senha criptografada que acabou de ser computada, o *login* é permitido, caso contrário é recusado.

A autenticação por senha é fácil de derrotar. Lemos com freqüência sobre grupos de alunos de faculdade, ou mes-

mo do segundo grau, que com a ajuda de seus confiáveis computadores domésticos simplesmente invadem algum sistema secreto de alto nível de uma grande corporação ou de um órgão do governo. Praticamente todo o tempo gasto em uma invasão consiste em adivinhar uma combinação de nome de usuário e de senha.

Embora estudos mais recentes tenham sido feitos (p. ex., Klein, 1990), o trabalho clássico sobre segurança por senhas continua sendo o feito por Morris e Thompson (1979) para sistemas UNIX. Eles compilaram uma lista de senhas possíveis: nomes e sobrenomes, nomes de rua, nomes de cidade, palavras de um dicionário de tamanho médio (também palavras soletradas de trás para a frente), números de placas de licença e cadeias curtas de caracteres aleatórios.

Eles, então, criptografaram cada um desses, utilizando o algoritmo conhecido de criptografia de senha e verificaram se qualquer das entradas de senhas criptografadas coincidia com sua lista. Mais de 86% de todas as senhas caíram em sua lista.

Se todas as senhas consistissem em 7 caracteres escolhidos aleatoriamente dos 95 caracteres ASCII imprimíveis, o espaço de pesquisa iria tornar-se 95^7, que é aproximadamente 7×10^{13}. À velocidade de 1.000 criptografias por segundo, levaria 2.000 anos para construir a lista contra a qual se poderia verificar um arquivo de senhas. Além disso, a lista preencheria 20 milhões de fitas magnéticas. Mesmo impondo que as senhas contenham pelo menos um caractere em letras minúsculas, um caractere em letras maiúsculas e um caractere especial e tenham pelo menos sete ou oito caracteres de comprimento seria uma melhora importante em relação às senhas irrestritas escolhidas pelos usuários.

Mesmo se for considerado politicamente impossível solicitar que os usuários selecionem senhas razoáveis, Morris e Thompson descreveram uma técnica que deixa o próprio ataque (criptografar um número grande de senhas de antemão) quase inútil. Sua idéia é associar um número aleatório de n bits a cada senha. O número aleatório é alterado sempre que a senha é alterada. O número aleatório é armazenado no arquivo de senha na forma não-criptografada, de modo que todo o mundo possa lê-lo. Em vez de simplesmente armazenar a senha criptografada no arquivo de senha, a senha e o número aleatório são primeiro concatenados e, então, criptografados juntos. Esse resultado criptografado é armazenado no arquivo de senhas.

Agora considere as implicações para um intruso que queira acumular uma lista de senhas possíveis, criptografá-las e salvar o resultado em um arquivo classificado, f, de modo que qualquer senha criptografada possa ser pesquisada facilmente. Se um intruso suspeita que *Marilyn* talvez tenha uma senha, não é mais suficiente apenas criptografar *Marilyn* e colocar o resultado em f. Ele terá de criptografar 2^n *strings*, como *Marilyn0000*, *Marilyn0001*, *Marilyn0002*, etc., e inserir todas elas em f. Essa técnica aumenta o tamanho de f por 2^n. O UNIX utiliza esse método com $n = 12$. É conhecido como **salgar** o arquivo de senhas. Algumas versões do UNIX tornam ilegível o próprio arquivo de senhas, mas oferecem um programa para pesquisar entradas no momento da solicitação, adicionando retardo suficiente para reduzir significativamente a velocidade do trabalho de qualquer intruso.

Embora esse método ofereça proteção contra intrusos que tentarem previamente computar uma lista grande de senhas criptografados, ele faz pouco para proteger um usuário *David* cuja senha é também *David*. Uma maneira de encorajar as pessoas a selecionar melhores senhas é fazer o computador oferecer esse aconselhamento. Alguns computadores têm um programa que gera palavras sem sentido aleatórias fáceis de pronunciar, como *fotalmente*, *lixeria* ou *bipedade* que você pode utilizar como senhas (preferivelmente com alguma letra maiúscula e caracteres especiais no meio).

Outros computadores exigem que os usuários alterem suas senhas regularmente, limitando o estrago feito se uma senha vazar. A forma mais extrema dessa abordagem é a **senha de uma vez**. Quando senhas de uma vez são utilizadas, o usuário recebe um livro que contém uma lista de senhas. Cada *login* utiliza a próxima senha na lista. Se um intruso vier a descobrir uma senha, ele não fará muito com ela, uma vez que da próxima vez uma senha diferente deverá ser utilizada. Recomenda-se que o usuário tente evitar perder o livro de senhas.

Também é óbvio que, quando uma senha está sendo digitada, o computador não deve exibir os caracteres digitados, para ocultá-los de olhos bisbilhoteiros próximos do terminal. O que é menos óbvio é que as senhas nunca devem ser armazenadas no computador na forma não-criptografada. Além disso, nem mesmo o CPD deveria guardar cópias não-criptografadas. Armazenar senhas não-criptografadas é procurar problemas.

Uma variação na idéia de senha é oferecer a cada novo usuário uma longa lista de perguntas e de respostas que, então, são armazenadas no computador na forma criptografada. As perguntas devem ser escolhidas de modo que o usuário não necessite escrevê-las. Perguntas típicas são:

1. Quem é a irmã de Tânia?
2. Em que rua ficava sua escola primária?
3. O professor Gustavo dava aulas de quê?

No momento do *login*, o computador solicita uma delas aleatoriamente e verifica a resposta.

Outra variação é a **resposta a desafio**. Quando isso é utilizado, o usuário seleciona um algoritmo quando se inscreve como um usuário, x^2, por exemplo. Quando o usuário estiver conectado, o computador digita um argumento, digamos 7, caso em que o usuário digita 49. O algoritmo pode ser diferente de manhã e à tarde, em dias diferentes da semana, em terminais diferentes e assim por diante.

Identificação Física

Uma abordagem completamente diferente para autorização é verificar se o usuário tem algum item, normalmente um cartão de plástico com uma tarja magnética. O

cartão é inserido no terminal, que, então, verifica de quem é esse cartão. Esse método pode ser combinado com uma senha; então, um usuário somente pode conectar-se se ele (1) tiver o cartão e (2) souber a senha. Os caixas automáticos de bancos geralmente funcionam dessa maneira.

Outra abordagem é medir as características físicas que são difíceis de falsificar. Por exemplo, uma impressão digital ou um leitor de voz no terminal poderia verificar a identidade do usuário. (A pesquisa será mais rápida se o usuário informar ao computador quem ele é, em vez de fazer o computador comparar a impressão digital dada em todo o banco de dados.) Reconhecimento visual direto ainda não é praticável, mas um dia pode vir a ser.

Outra técnica é a análise da assinatura. O usuário assina seu nome com uma caneta especial conectada ao terminal e o computador compara-o *on-line* a uma amostra conhecida e armazenada. Melhor ainda é comparar não a assinatura, mas sim os movimentos da caneta enquanto ela está sendo escrita. Um bom falsificador pode ser capaz de copiar a assinatura, mas não terá uma pista da ordem exata em que os movimentos foram feitos.

A análise do comprimento dos dedos é surpreendentemente prática. Quando isso é utilizado, cada terminal tem um dispositivo como o da Figura 5-21. O usuário insere a sua mão nele, e o comprimento de todos seus dedos é medido e verificado contra o banco de dados.

Poderíamos prosseguir com mais exemplos, porém dois ajudarão a tornar mais claro um ponto importante. Os gatos e outros animais marcam seu território urinando no seu perímetro. Aparentemente gatos podem identificar-se dessa maneira. Suponha que alguém apareça com um dispositivo minúsculo capaz de fazer uma análise da urina instantânea, oferecendo assim uma identificação à prova de falhas. Cada terminal poderia ser equipado com um desses dispositivos, junto com um discreto aviso dizendo: "Para *login*, por favor deposite a amostra aqui". Talvez esse seja um sistema absolutamente inquebrável, mas provavelmente teria um sério problema de aceitação por parte do usuário.

O mesmo poderia ser dito de um sistema consistindo em um coletor e em um pequeno espectrógrafo. O usuário seria solicitado a pressionar seu polegar contra o coletor, extraindo assim uma gota de sangue para análise espectrográfica. O ponto é que qualquer esquema de autenticação deve ser psicologicamente aceitável para a comunidade de usuários. As medidas do comprimento do dedo provavelmente não causarão qualquer problema, mas mesmo algo tão pouco indiscreto ou inconveniente como armazenar impressões digitais *on-line* pode ser inaceitável para muitas pessoas.

Contramedidas

As instalações de computador que são realmente sérias quanto à segurança, algo que, com freqüência, acontece no dia seguinte depois que um invasor quebrou a segurança e fez um estrago importante, freqüentemente adotam passos para tornar uma entrada não-autorizada muito mais difícil. Por exemplo, cada usuário poderia ter permissão para conectar-se somente a partir de um terminal específico e apenas durante certos dias da semana e em certas horas do dia.

Linhas de telefone discadas poderiam funcionar da seguinte maneira. Qualquer pessoa pode discar e conectar-se, mas após um *login* bem-sucedido, o sistema imediata-

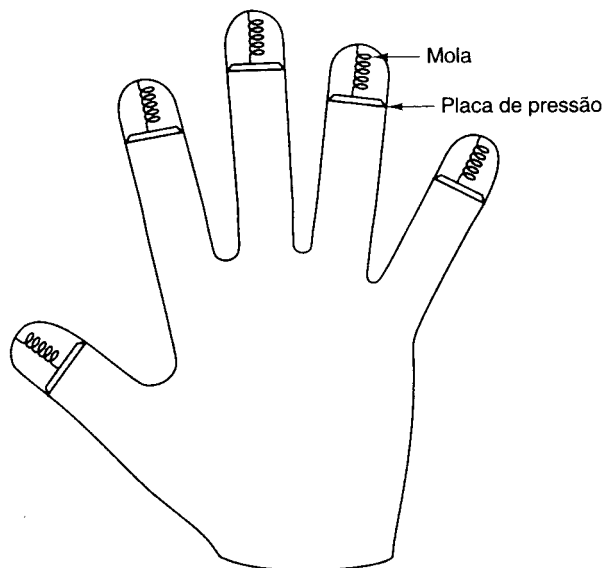



Figura 5-21 Um dispositivo para medir comprimento dos dedos.

mente derruba a conexão e chama de volta o usuário em um número previamente definido. Essa medida significa que um intruso não pode quebrar a segurança a partir de qualquer linha de telefone, mas sim apenas a partir do telefone (de casa) do usuário. Em qualquer caso, com ou sem retorno de chamada, o sistema deve levar pelo menos 10 segundos para verificar qualquer senha digitada em uma linha discada e deve aumentar esse tempo após várias tentativas de *login* consecutivas malsucedidas, para reduzir a velocidade das tentativas do intruso. Após três tentativas falhas de *login*, a linha deve ser desconectada por 10 minutos, e o pessoal de segurança notificado.

Todos os *logins* devem ser registrados. Quando um usuário efetua *login*, o sistema deve informar o momento e o terminal do *login* anterior, assim ele pode detectar uma possível invasão.

O próximo passo é estabelecer armadilhas na forma de iscas para capturar intrusos. Um esquema simples é ter um nome especial de *login* com uma senha fácil (p. ex., nome de *login*: *guest*, senha: *guest*). Sempre que qualquer pessoa conectar-se, utilizando esse nome, os especialistas de segurança de sistema imediatamente são notificados. Outras armadilhas podem ser *bugs* "fáceis de achar" no sistema operacional e coisas semelhantes, projetados com o propósito de capturar intrusos no ato. Stoll (1989) escreveu um relato divertido das armadilhas que ele montou para rastrear um espião que invadiu um computador de uma universidade, procurando segredos militares.

5.5 MECANISMOS DE PROTEÇÃO

Nas seções anteriores, vimos muitos problemas potenciais, algum deles técnicos e alguns não. Nas seções a seguir, vamos concentrar-nos em algumas técnicas detalhadas que são utilizadas nos sistemas operacionais para proteger arquivos e outras coisas. Todas essas técnicas fazem uma distinção clara entre política (quais dados devem ser protegidos de quem) e mecanismo (como o sistema impõe a política). A separação entre política e mecanismo é discutida em (Levin *et al.*, 1975). Nossa ênfase estará no mecanismo, não na política. Para material mais avançado, veja (Sandhu, 1993).

Em alguns sistemas, a proteção é imposta por um programa chamado **monitor de referência**. Cada vez que é tentado um acesso a um recurso potencialmente protegido, o sistema primeiro exige que o monitor de referência verifique sua legalidade. O monitor de referência, então, olha em suas tabelas de políticas e toma uma decisão. A seguir, descreveremos o ambiente em que um monitor de referência opera.

5.5.1 Domínios de Proteção

Um sistema de computador contém muitos "objetos" que precisam ser protegidos. Esses objetos podem ser hardware (p. ex., CPUs, segmentos de memória, unidades de disco ou impressoras) ou podem ser software (p. ex., processos, arquivos, bancos de dados ou semáforos).

Cada objeto tem um nome único por meio do qual ele é referenciado e um conjunto limitado de operações que os processos têm permissão para executar. As operações READ e WRITE são apropriadas para um arquivo; UP e DOWN fazem sentido em um semáforo.

É óbvio que é necessário um meio de proibir que processos acessem objetos a que eles não têm acesso autorizado. Além disso, esse mecanismo também deve tornar possível restringir os processos a um subconjunto das operações legais quando isso for necessário. Por exemplo, o processo *A* pode ser autorizado a ler, mas não a gravar, o arquivo *F*.

Para discutir diferentes mecanismos de proteção, é útil apresentar o conceito de domínio. Um **domínio** é um conjunto de pares (objeto, direitos). Cada par especifica um objeto e algum subconjunto das operações que podem ser executadas nele. Um **direito** nesse contexto de permissão significa executar uma das operações.

A Figura 5-22 mostra três domínios, mostrando os objetos em cada domínio e os direitos (Read, Write eXecute – leitura, escrita, execução) disponíveis para cada objeto. Note que a *Impressora1* está em dois domínios ao mesmo tempo. Embora não mostrado nesse exemplo, é possível o mesmo objeto estar em múltiplos domínios, com *diferentes* direitos em cada um.

A cada instante de tempo, cada processo executa em algum domínio de proteção. Em outras palavras, há uma coleção de objetos que ele pode acessar, e para cada objeto há um conjunto de direitos. Os processos também podem alternar de domínio para domínio durante a execução. As regras para comutação de domínio são bastante dependentes do sistema.

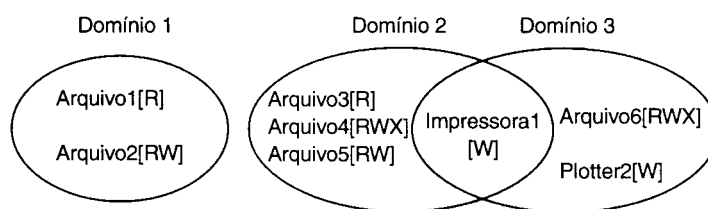



Figura 5-22 Três domínios de proteção.

Para dar uma idéia mais concreta de domínio de proteção, vejamos o UNIX. No UNIX, o domínio de um processo é definido por seu *uid* e seu *gid*. Dada qualquer combinação (*uid*, *gid*), é possível fazer uma lista completa de todos os objetos (arquivos, incluindo dispositivos de E/S representados por arquivos especiais, etc.) que podem ser acessados e se eles podem ser acessados para leitura, para gravação ou para execução. Dois processos com a mesma combinação (*uid*, *gid*) terão acesso exatamente ao mesmo conjunto de objetos. Processos com valores (*uid*, *gid*) diferentes terão acesso a um conjunto diferente de arquivos, embora haja considerável sobreposição na maioria dos casos.

Além disso, cada processo no UNIX tem duas metades: a parte do usuário e a parte do *kernel*. Quando o processo faz uma chamada de sistema, ele alterna da parte do usuário para a parte do *kernel*. A parte do *kernel* tem acesso a um conjunto de objetos diferente da parte do usuário. Por exemplo, o *kernel* pode acessar todas as páginas na memória física, o disco inteiro e todos os outros recursos protegidos. Assim, uma chamada de sistema causa uma comutação de domínio.

Quando um processo faz um EXEC em um arquivo com o bit SETUID ou SETGID ativado, ele adquire um novo *uid* ou um *gid* efetivo. Com uma combinação (*uid*, *gid*) diferente, ele tem um conjunto diferente de arquivos e de operações disponíveis. A execução de um programa com SETUID ou SETGID é também uma comutação de domínio, uma vez que os direitos disponíveis agora são diferentes.

Uma questão importante é como o sistema monitora quais objetos pertencem a qual domínio. Conceitualmente, pelo menos, pode-se conceber uma grande matriz, com as linhas sendo os domínios, e as colunas sendo os objetos. Cada elemento da matriz lista os direitos, se houver algum, que o domínio contém para o objeto. A matriz para a Figura 5-22 é mostrada na Figura 5-23. Dados essa matriz e o número atual de domínio, o sistema pode dizer se é permitido um acesso a um dado objeto de uma maneira particular a partir de um domínio especificado.

A própria comutação de domínio pode facilmente ser incluída no modelo de matriz, percebendo que o domínio em si é um objeto, com a operação ENTER. A Figura 5-24 mostra a matriz da Figura 5-23 novamente, só que agora com os três domínios como os objetos em si. Os processos no domínio 1 podem alternar para o domínio 2, mas uma vez lá, não podem voltar. Essa situação modela a execução de um programa SETUID no UNIX. Nenhuma outra alternação de domínio é permitida nesse exemplo.

5.5.2 Listas de Controle de Acesso

Na prática, o armazenamento da matriz da Figura 5-24 raramente é feito, porque a matriz é grande e esparsa. A

Objeto

Domínio	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter2
1	Leitura	Leitura Escrita						
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita	
3						Leitura Escrita Execução	Escrita	Escrita

Figura 5-23 Uma matriz de proteção.

Objeto

Domínio	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter2	Domínio1	Domínio2	Domínio3
1	Leitura	Leitura Escrita								Enter	
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita				
3						Leitura Escrita Execução	Escrita	Escrita			

Figura 5-24 Uma matriz de proteção com domínios como objetos.

maioria dos domínios não tem nenhum acesso para a maioria dos objetos; então, armazenar uma matriz grande e na sua maior parte vazia é um desperdício de espaço em disco. Dois métodos que são práticos, entretanto, são armazenar a matriz por linhas ou por colunas e, então, armazenar somente os elementos não-vazios. As duas abordagens são surpreendentemente diferentes. Nesta seção, veremos armazenamento por coluna; na seguinte, estudaremos armazenamento por linha.

A primeira técnica consiste em associar a cada objeto uma lista (ordenada), contendo todos os domínios que podem acessar o objeto, e como. Essa lista é chamada **lista de controle de acesso** ou (**ACL –** ***Access List Control***). Se fosse implementada no UNIX, a maneira mais fácil seria colocar a ACL para cada arquivo em um bloco separado de disco e incluir o número deste bloco no nó-i do arquivo. Como somente as entradas não-vazias da matriz são armazenadas, o armazenamento total exigido para todas as ACLs combinadas é muito menor que a que seria necessária para a matriz inteira.

Como um exemplo de como as ACLs funcionam, vamos continuar imaginando que elas sejam utilizadas no UNIX, onde um domínio é especificado por um par (*uid, gid*). Realmente, as ACLs foram utilizadas no modelo do UNIX, o MULTICS, mais ou menos na maneira como descreveremos, então, o exemplo não é tão hipotético.

Vamos agora supor que temos quatro usuários (i. e., *uids*) *Jan*, *Els*, *Jelle* e *Maaike*, que pertencem aos grupos *sistema*, *staff*, *estudante* e *estudante*, respectivamente. Suponha que alguns arquivos têm as seguintes ACLs:

Arquivo0: (Jan, *, RWX)
Arquivo1: (Jan, sistema, RWX)
Arquivo2: (Jan, *, RW-), (Els, staff, RW-), (Maaike, *, RW-)
Arquivo3: (*, estudante, R- -)
Arquivo4: (Jelle, *, - - -), (*, estudante, R- -)

Cada entrada de ACL, entre parênteses, especifica um *uid*, um *gid* e os acessos permitidos (Read, Write, eXecute – RWX). Um asterisco significa todos os *uids* ou *gids*. *Arquivo0* poder ser lido, gravado ou executado por qualquer processo com *uid* = *Jan* e qualquer *gid*. *Arquivo1* poder ser acessado somente por processos com *uid* = *Jan* e *gid* = *sistema*. Um processo que tem *uid* = *Jan* e *gid* = *staff* pode acessar *Arquivo0*, mas não *Arquivo1*. *Arquivo2* poder ser lido ou gravado por processos com *uid* = *Jan* e qualquer *gid*, lido por processos com *uid* = *Els* e *gid* = *staff*; ou por processos com *uid* = *Maaike* e qualquer *gid*. *Arquivo3* poder ser lido por qualquer aluno. *Arquivo4* é especialmente interessante. Ele diz que qualquer pessoa com *uid* = *Jelle*, em qualquer grupo, não tem absolutamente nenhum acesso, mas todos os outros alunos podem lê-lo. Utilizando ACLs, é possível proibir *uids* ou *gids* específicos de acessar um objeto, enquanto permite todas as outras pessoas na mesma classe.

Bem, já falamos demais sobre o que o UNIX não faz. Agora vejamos o que ele *realmente* faz. Ele oferece três bits, *rwx*, por arquivo para o proprietário, o grupo do proprietário e os outros. Esse esquema é simplesmente a ACL novamente, mas compactada para 9 bits. Trata-se de uma lista associada com o objeto, dizendo quem pode acessá-la e como. Embora o esquema de 9 bits do UNIX seja nitidamente menos geral que um sistema de ACL pleno, na prática ele é adequado e sua implementação é muito mais simples e barata.

O proprietário de um objeto pode alterar sua ACL a qualquer momento, tornando assim fácil proibir acessos que anteriormente eram permitidos. O único problema é que alterar a ACL mais provavelmente não afetará qualquer usuário que atualmente esteja utilizando o objeto (p. ex., que atualmente tem o arquivo aberto).

5.5.3 Capacidades

A outra maneira de cortar em fatias a matriz da Figura 5-24 é por linhas. Quando esse método é utilizado, associado com cada processo está uma lista de objetos que podem ser acessados, junto com uma indicação de quais operações são permitidas em cada um, em outras palavras, seu domínio. Essa lista é chamada **lista de capacitação**, e os itens individuais dentro dela são chamados **capacidades** (Dennis e Van Horn, 1966; Fabry, 1974).

Uma típica lista de capacitação é mostrada na Figura 5-25. Cada capacidade tem um campo *Tipo*, que diz qual é o tipo de um objeto, um campo *Direitos*, que é um mapa de bits, indicando quais das operações legais são permitidas para esse tipo de objeto e um campo *Objeto*, que é um ponteiro para o próprio objeto (p. ex., seu número de nó-i). As listas de capacitação são elas próprias objetos e podem ser apontadas por outra lista de capacitação, facilitando assim o compartilhamento de subdomínios. As capacidades são freqüentemente referidas por sua posição na lista de capacitação. Um processo poderia dizer: "Leia 1K do arquivo apontado por capacidade 2". Essa forma de endereçamento é semelhante a utilizar descritores de arquivo no UNIX.

É relativamente óbvio que as listas de capacitações ou **listas C** como freqüentemente são chamadas, devem ser protegidas contra alterações indevidas por parte do usuário. Três métodos foram propostos para protegê-las. A primeira maneira requer uma **arquitetura etiquetada**, um projeto de hardware em que cada palavra de memória tem um bit extra (ou etiqueta) que diz se a palavra contém uma capacidade ou não. O bit de etiqueta não é utilizado por instruções aritméticas, de comparação, nem outras funções comuns semelhantes e pode ser modificado somente por programas que executam no modo *kernel* (i. e., o sistema operacional).

A segunda maneira é manter a lista C dentro do sistema operacional e simplesmente fazer os processos referir as capacidades por seu número de entrada, como mencionado anteriormente. O Hydra (Wulf *et al.*, 1974) trabalhava dessa maneira.

A terceira maneira é manter a lista C no espaço do usuário, mas criptografar cada capacidade com uma chave secreta desconhecida para o usuário. Essa abordagem é

#	Tipo	Direitos	Objeto
0	File	R--	Ponteiro para Arquivo3
1	File	RWX	Ponteiro para Arquivo4
2	File	RW-	Ponteiro para Arquivo5
3	Pointer	-W-	Ponteiro para Impressora1

Figura 5-25 A lista de capacitação para o domínio 2 na Figura 5-23.

particularmente adequada para sistemas distribuídos e é utilizada extensamente pelo Amoeba (Tanenbaum *et al.*, 1990).

Além dos direitos específicos dependentes do objeto, como leitura e execução, as capacidades normalmente têm **direitos genéricos** que são aplicáveis a todos os objetos. Exemplos de direitos genéricos são

1. A capacidade de cópia: cria uma nova capacidade para o mesmo objeto.
2. Copiar objeto: cria um objeto duplicado com uma nova capacidade.
3. Remover capacidade: exclui uma entrada da lista C; não afeta o objeto.
4. Destruir objeto: remove permanentemente um objeto e uma capacidade.

Uma última observação que merece ser feita sobre sistemas de capacitação é que revogar acesso a um objeto é bastante difícil. É difícil para o sistema localizar todas as capacidades destacadas para qualquer objeto e tomá-las de volta, uma vez que elas podem estar armazenadas em listas C por todo o disco. Uma abordagem é ter cada capacidade apontando para um objeto indireto, em vez de para o próprio objeto. Por ter o objeto indireto apontando para o objeto real, o sistema sempre pode quebrar essa conexão, invalidando, assim, as capacidades. (Quando uma capacidade para o objeto indireto posteriormente é apresentada ao sistema, o usuário descobrirá que o objeto indireto agora está apontando para um objeto nulo.)

Outra maneira de obter revogação é o esquema utilizado no Amoeba. Cada objeto contém um número aleatório longo, que também está presente na capacidade. Quando uma capacidade é apresentada para utilização, os dois são comparados. Somente quando estão de acordo é que as operações são permitidas. O proprietário de um objeto pode solicitar que o número aleatório no objeto seja alterado, invalidando, dessa forma, capacidades existentes. Nenhum esquema permite revogação seletiva, isto é, tomar de volta, digamos, apenas a permissão de um usuário em particular.

5.5.4 Canais Secretos

Mesmo com listas de controle de acesso e de capacitações, podem ocorrer vazamentos na segurança. Nesta seção, discutimos uma classe de problema. Tais idéias devem-se a Lampson (1973).

O modelo de Lampson envolve três processos e é principalmente aplicável a sistemas de compartilhamento de tempo de grande porte. O primeiro processo é o cliente, o qual deseja que algum trabalho seja realizado pelo segundo, o servidor. O cliente e o servidor não confiam inteiramente um no outro. Por exemplo, o trabalho do servidor é ajudar clientes no preenchimento de seus formulários de imposto. Os clientes estão preocupados com que o servidor registre secretamente seus dados financeiros, por exemplo, mantendo uma lista secreta de quem ganha quanto e, então, vender a lista. O servidor está preocupado com o fato de os clientes poderem tentar roubar o valioso programa de imposto.

O terceiro processo é o colaborador, que, de fato, está conspirando com o servidor para roubar os dados confidenciais do cliente. O colaborador e o servidor geralmente são possuídos pela mesma pessoa. Esses três processos são mostrados na Figura 5-26. O objeto desse exercício é projetar um sistema em que é impossível para o servidor vazar para o colaborador as informações que legitimamente recebeu do cliente. Lampson chamou isso de **problema do confinamento**.

Do ponto de vista do projetista de sistema, o objetivo é encapsular ou confinar o servidor de tal maneira que ele não possa passar as informações para o colaborador. Utilizando um esquema de matriz de proteção, podemos facilmente garantir que o servidor não possa comunicar-se com o colaborador por meio da gravação de um arquivo a que o colaborador tenha acesso de leitura. Provavelmente, também podemos assegurar que o servidor não possa comunicar-se com o colaborador, utilizando o mecanismo de comunicação interprocessos do sistema.

Infelizmente, podem estar disponíveis canais mais sutis de comunicação. Por exemplo, o servidor pode tentar comunicar um fluxo binário de bits como segue. Para enviar um bit 1, ele trabalha o máximo que pode por um intervalo fixo de tempo. Para enviar um bit 0, ele vai dormir pelo mesmo intervalo de tempo.

O colaborador pode tentar detectar o fluxo de bits por meio do cuidadoso monitoramento do seu tempo de resposta. Em geral, ele obterá melhor resposta quando o servidor estiver enviando um 0 do que quando o servidor estiver enviando um 1. Esse canal de comunicação é conhecido como um **canal secreto** (*covert channel*) e é ilustrado na Figura 5-26(b)

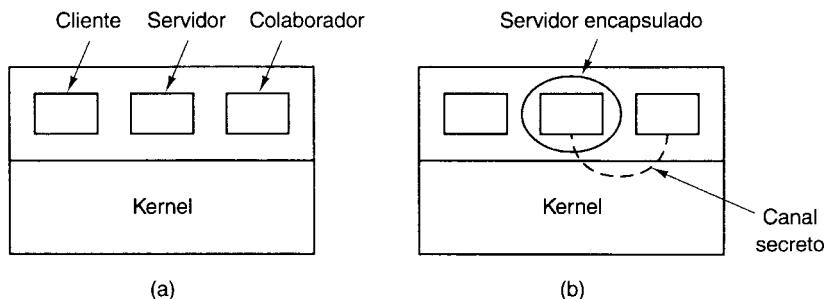



Figura 5-26 (a) Os processos do cliente, do servidor e do colaborador. (b) O servidor encapsulado ainda pode vazar para o colaborador via canais secretos.

Naturalmente, o canal secreto é um canal com muito ruído, contendo muitas informações estranhas, mas as informações de confiança podem ser enviadas por um canal ruidoso, utilizando um código para correção de erros (p. ex., um código de Hamming ou algo até mais sofisticado). O uso de um código para correção de erros reduz a largura de banda, já baixa, do canal secreto ainda mais, mas ainda pode ser suficiente para vazar informações substanciais. É relativamente óbvio que nenhum modelo de proteção baseado em uma matriz de objetos e de domínios impedirá esse tipo de vazamento.

Modular o uso da CPU não é o único canal secreto. A taxa de paginação também pode ser modulada (muitas falhas de página para um 1, nenhuma falha de página para um 0). De fato, quase qualquer maneira de degradar o desempenho do sistema de modo temporizado é candidata. Se o sistema oferecer uma maneira de bloquear arquivos, então, o servidor pode bloquear algum arquivo para indicar um 1, e desbloquear para indicar um 0. Em alguns sistemas, é possível um processo detectar o status de um bloqueio mesmo em um arquivo que ele não pode acessar.

Adquirir e liberar recursos dedicados (unidades de fita, plotadoras, etc.) também pode ser utilizado para sinalizar. O servidor adquire o recurso para enviar um 1 e libera-o para enviar um 0. No UNIX, o servidor poderia criar um arquivo para indicar um 1 e removê-lo para indicar um 0: o colaborador poderia utilizar a chamada de sistema ACCESS para ver se o arquivo existe. Essa chamada funciona mesmo que o colaborador não tenha nenhuma permissão para utilizar o arquivo. Infelizmente existem muitos outros canais secretos.

Lampson também menciona uma maneira de vazar as informações para o proprietário (humano) do processo de servidor. Presumivelmente o processo de servidor será intitulado a dizer a seu proprietário quanto trabalho fez em favor do cliente; então, o cliente pode ser cobrado. Se a conta real do cálculo for, digamos, 100 dólares e a renda do cliente é 53K de dólares os servidores poderiam informar a conta como 100,53 para seu proprietário.

Simplesmente localizar todos os canais secretos, deixando de lado a ação de bloqueá-los, é extremamente difícil. Na prática, há pouco que pode ser feito. Introduzir um processo que causa falhas de página aleatoriamente ou, de outro modo, que gasta seu tempo degradando o desempenho do sistema a fim de reduzir a largura de banda dos canais secretos não é uma proposta atraente.

5.6 VISÃO GERAL DO SISTEMA DE ARQUIVOS DO MINIX

Como qualquer sistema de arquivos, o sistema de arquivos do MINIX deve lidar com todas as questões que acabamos de estudar. Ele deve alocar e desalocar espaço para arquivos, monitorar blocos de disco e liberar espaço, oferecer alguma maneira de proteger arquivos contra uso não-autorizado e assim por diante. No restante deste capítulo, vamos aprofundar-nos no MINIX para ver como ele realiza esses objetivos.

Na primeira parte deste capítulo, repetidamente referimo-nos ao UNIX em vez de ao MINIX por generalidade, embora a interface externa dos dois seja praticamente idêntica. Agora nos concentraremos no projeto interno do MINIX. Para as informações sobre aspectos internos do UNIX, veja Thompson (1978), Bach (1987), Lions (1996) e Vahalia (1996).

O sistema de arquivos do MINIX é somente um grande programa em C que executa no espaço do usuário (veja a Figura 2-26). Para ler e para gravar arquivos, os processos de usuário enviam mensagens para o sistema de arquivos dizendo o que eles querem que seja feito. O sistema de arquivos faz o trabalho e, então, envia de volta uma resposta. O sistema de arquivos é, de fato, um servidor de arquivos de rede que está executando na mesma máquina que o chamador.

Esse projeto tem algumas implicações importantes. De um lado, o sistema de arquivos pode ser modificado, experimentado e testado quase completamente independente do restante do MINIX. Por outro, é muito fácil mover o sistema de arquivos inteiro para qualquer computador que tenha um compilador C, compilá-lo aí e utilizá-lo como um servidor remoto independente de arquivos UNIX. As únicas

alterações que precisam ser feitas estão na área de como as mensagens são enviadas e recebidas, que difere de sistema para sistema.

Nas seções a seguir, apresentaremos uma visão geral de muitas das áreas-chaves do projeto do sistema de arquivos. Especificamente, veremos as mensagens, o arranjo do sistema de arquivos, os mapas de bits, nós-i, o *cache* de blocos, os diretórios e caminhos, os descritores de arquivo, o bloqueio de arquivo e os arquivos especiais (mais canalizações). Depois que estudarmos todos esses temas, mostraremos um exemplo simples de como os pedaços ajustam-se entre si, rastreando o que acontece quando um processo de usuário executa a chamada de sistema READ.

5.6.1 Mensagens

O sistema de arquivos aceita 39 tipos de mensagens que solicitam trabalho. Todas exceto duas são para chamadas de sistema MINIX. As duas exceções são mensagens geradas por outras partes do MINIX. Das chamadas de sistema, 31 são aceitas de processos de usuário. Seis mensagens de chamada de sistema são para chamadas de sistema tratadas primeiro pelo gerenciador de memória que, então, chama o sistema de arquivos para fazer uma parte do trabalho. Duas outras mensagens também são processadas pelo sistema de arquivos. As mensagens são mostradas na Figura 5-27.

A estrutura do sistema de arquivos é basicamente a mesma do gerenciador de memória e de todas as tarefas de E/S. Ele tem um laço principal que espera uma mensagem chegar; quando uma mensagem chega, seu tipo é extraído e utilizado como um índice em uma tabela de ponteiros que contém os procedimentos dentro do sistema de arquivos que tratam todos os tipos. Então, o procedimento apropriado é chamado, faz seu trabalho e retorna um valor de status. O sistema de arquivos, então, envia de volta uma resposta ao chamador e volta para o topo do laço, esperando a próxima mensagem.

5.6.2 Arranjo do Sistema de Arquivos

Um sistema de arquivos MINIX é uma entidade lógica autocontida com nós-i, com diretórios e com blocos de dados. Pode ser armazenado em qualquer dispositivo de bloco, como um disquete ou (parte de) um disco rígido. Em todos os casos, o arranjo do sistema de arquivos tem a mesma estrutura. A Figura 5-28 mostra esse arranjo para um disquete de 360K com 128 nós-i e um tamanho de bloco de 1K. Sistemas de arquivos maiores, ou aqueles com mais ou menos nós-i ou um tamanho de bloco diferente, terão os mesmos seis componentes na mesma ordem, mas seus tamanhos relativos podem ser diferentes.

Cada sistema de arquivos começa com um **bloco de inicialização**. Esse contém código executável. Quando o computador é ligado, o hardware lê o bloco de inicialização do dispositivo de inicialização para a memória, salta para ele e começa a executar seu código. O código do bloco de inicialização começa o processo de carregamento do sistema operacional em si. Uma vez que o sistema foi inicializado, o bloco de inicialização não é mais utilizado. Nem toda unidade de disco pode ser utilizada como um dispositivo de inicialização, mas mantendo a estrutura uniforme, cada dispositivo de blocos tem um bloco reservado para o código do bloco de inicialização. Na pior das hipóteses, essa estratégia desperdiça um bloco. Para impedir que o hardware tente inicializar um dispositivo não-inicializável um **número mágico** é colocado em uma posição conhecida no bloco de inicialização quando e somente quando o código executável é gravado no dispositivo. Quando inicializa de um dispositivo, o hardware (na realidade, o código de BIOS) irá recusar-se a tentar carregar de um dispositivo em que falta o número mágico. Fazendo isso, previne-se que lixo seja inadvertidamente utilizado como um programa de inicialização.

O **superbloco** contém as informações que descrevem o arranjo do sistema de arquivos. Ele é ilustrado na Figura 5-29. A principal função do superbloco é dizer ao sistema de arquivos o tamanho dos vários pedaços que compõem tal sistema. Dados o tamanho de bloco e o número de nós-i, é fácil calcular o tamanho do mapa de bits de nós-i e o número de blocos de nós-i. Por exemplo, para um bloco de 1K, cada bloco do mapa de bits tem 1K bytes (8K bits) e assim podem monitorar o status de até 8192 nós-i. (Realmente o primeiro bloco pode tratar somente até 8191 nós-i, uma vez que não há um nó-i 0 (zero), mas é fornecido um bit no mapa de bits, de qualquer maneira.) Para 10.000 nós-i, dois blocos de mapa de bits são necessários. Uma vez que cada nó-i ocupa 64 bytes, um bloco de 1K armazena até 16 nós-i. Com 128 nós-i utilizáveis, 8 blocos de disco são necessários para conter todos.

Explicaremos a diferença entre zonas e blocos detalhadamente mais adiante, mas por enquanto é suficiente dizer que o espaço em disco pode ser alocado em unidades (zonas) de 1, 2, 4, 8 ou em geral 2^n blocos. O mapa de bits de zonas monitora o espaço livre em zonas, não em blocos. Para todos os disquetes-padrão utilizados pelo MINIX, os tamanhos de blocos e de zonas são os mesmos (1K); portanto, para uma primeira abordagem, uma zona é o mesmo que um bloco nesses dispositivos. Até chegarmos aos detalhes da alocação de espaço mais adiante no capítulo, é adequado pensar "bloco" sempre que você ver "zona".

Note que o número de blocos por zona não é armazenado no superbloco, uma vez que ele nunca é necessário. Tudo o que é necessário é logaritmo de base 2 da zona para proporção de bloco, que é utilizado como contagem de deslocamento para converter blocos em zonas e vice-versa. Por exemplo, com 8 blocos por zona, $\log_2 8 = 3$, portanto, para localizar a zona que contém o bloco 128, deslocamos 128 para a direita 3 bits a fim de obter a zona 16.

O mapa de bits de zonas inclui somente as zonas de dados (i. e., os blocos utilizados para os mapas de bit e nós-i não estão no mapa), com a primeira zona de dados designada zona 1 no mapa de bits. Como com o mapa de bits

Mensagens dos usuários	Parâmetros de entrada	Valor da resposta
ACCESS	Nome do arquivo, modo de acesso	Status
CHDIR	Nome do novo diretório de trabalho	Status
CHMOD	Nome do arquivo, novo modo	Status
CHOWN	Nome do arquivo, novo proprietário, grupo	Status
CHROOT	Nome do novo diretório-raiz	Status
CLOSE	Descritor de arquivo do arquivo a ser fechado	Status
CREAT	Nome do arquivo a ser criado, modo	Descritor de arquivo
DUP	Descritor de arquivo (para dup2, dois fds)	Novo descritor de arquivo
FCNTL	Descritor de arquivo, código de função, arg	Depende da função
FSTAT	Nome do arquivo, buffer	Status
IOCTL	Descritor de arquivo, código de função, arg	Status
LINK	Nome do arquivo a vincular, nome do vínculo	Status
LSEEK	Descritor de arquivo, deslocamento, de onde	Nova posição
MKDIR	Nome do arquivo, modo	Status
MKNOD	Nome de dir ou modo especial, endereço	Status
MOUNT	Arquivo especial, onde montar, sinalizador de ro	Status
OPEN	Nome do arquivo a abrir, sinalizador de r/w	Descritor de arquivo
PIPE	Ponteiro para 2 descritores de arquivo (modificado)	Status
READ	Descritor de arquivo, buffer, quantos bytes	# Bytes lidos
RENAME	Nome do arquivo, nome do arquivo	Status
RMDIR	Nome do arquivo	Status
STAT	Nome do arquivo, buffer de status	Status
STIME	Ponteiro para tempo atual	Status
SYNC	(Nenhum)	Sempre OK
TIME	Ponteiro para onde colocar o tempo atual	Status
TIMES	Ponteiro para buffer de tempos do processo e do filho	Status
UMASK	Complemento de máscara de modo	Sempre OK
UMOUNT	Nome do arquivo especial para desmontar	Status
UNLINK	Nome do arquivo para desvincular	Status
UTIME	Nome do arquivo, tempos do arquivo	Sempre OK
WRITE	Descritor do arquivo, buffer, quantos bytes	# Bytes gravados
Mensagens do MM	**Parâmetros de entrada**	**Valor da resposta**
EXEC	Pid	Status
EXIT	Pid	Status
FORK	Pid do pai, pid do filho	Status
SETGID	Pid, gid real e efetivo	Status
SETSID	Pid	Status
SETUID	Pid, uid real e efetivo	Status
Outras mensagens	**Parâmetros de entrada**	**Valor da resposta**
REVIVE	Processo a reanimar	(Nenhuma resposta)
UNPAUSE	Processo a verificar	(Veja texto)

Figura 5-27 As mensagens do sistema de arquivos. Os parâmetros do nome de arquivo são sempre ponteiros para o nome. O código de status como valor de resposta significa *OK* ou *ERROR*.

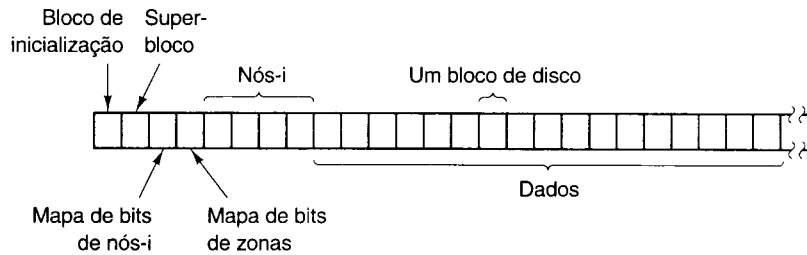



Figura 5-28 O arranjo de disco para o disco mais simples: um disquete de 360K, com 128 nós-i e um tamanho de bloco de 1K (i. e., dois setores de 512 bytes consecutivos são tratados como um único bloco).

de nós-i, o bit 0 no mapa não é utilizado; então, o primeiro bloco no mapa de bits de zonas pode mapear 8191 zonas e blocos subseqüentes podem mapear 8192 zonas cada um. Se examinar os mapas de bits em um disco recentemente formatado, você descobrirá que tanto os mapas de bit de nó-i como de zonas têm 2 bits configurados como 1. Um é para o nó-i 0 ou zona 0 (inexistentes); o outro é para o nó-i e a zona, utilizados pelo diretório-raiz no dispositivo, que é colocado aí quando o sistema de arquivos é criado.

As informações no superbloco são redundantes porque, às vezes, elas são necessárias de uma forma e, às vezes, de outra. Com 1K dedicado ao superbloco, faz sentido computar essas informações em todas as formas que sejam necessárias, em vez de ter de recomputá-las freqüentemente durante a execução. O número de zona da primeira zona de dados no disco, por exemplo, pode ser calculado a partir do tamanho de bloco, tamanho da zona, número de nós-i e número de zonas, mas é mais rápido simplesmente man-

Figura 5-29 O superbloco do MINIX.

tê-lo no superbloco. O restante do superbloco é desperdiçado de qualquer modo, portanto, utilizar outra palavra dele não custa nada.

Quando o MINIX é inicializado, o superbloco para o dispositivo-raiz é carregado em uma tabela na memória. De maneira semelhante, de acordo como outros sistemas de arquivos são montados, seus superblocos também são trazidos para a memória. A tabela do superbloco armazena alguns campos ausentes no disco. Esses incluem sinalizadores que permitem que o acesso a um dispositivo seja especificado como apenas de leitura ou como seguindo uma convenção de ordem de bytes oposta ao padrão e campos para acelerar o acesso, indicando pontos nos mapas de bits abaixo dos quais todos os bits são marcados como utilizados. Além disso, há um campo que descreve o dispositivo do qual o superbloco veio.

Antes de um disco poder ser utilizado como um sistema de arquivos MINIX, ele deve receber a estrutura de dados da Figura 5-28. O programa utilitário *mkfs* foi oferecido para construir sistemas de arquivos. Esse programa tanto pode ser chamado por um comando do tipo

```
mkfs /dev/fd1 1440
```

para construir um sistema de arquivos vazio de 1440 blocos no disquete na unidade 1, como pode receber um arquivo de protótipo, listando diretórios e arquivos para incluir no novo sistema de arquivos. Esse comando também coloca um número mágico no superbloco para identificar o sistema de arquivos como um sistema de arquivos MINIX válido. O sistema de arquivos MINIX desenvolveu-se e alguns aspectos do sistema de arquivos (p. ex., o tamanho dos nós-i) eram diferentes em versões anteriores. O número mágico identifica a versão de *mkfs* que criou o sistema de arquivos, de modo que as diferenças podem ser ajustadas. Tentativas de montar um sistema de arquivos que não no formato MINIX, como um disquete MS-DOS, serão rejeitadas pela chamada de sistema MOUNT, que verifica o superbloco para um número mágico válido e outras coisas.

5.6.3 Mapas de Bits

O MINIX monitora quais nós-i e quais zonas estão livres utilizando dois mapas de bits (veja Figura 5-29). Quando um arquivo é removido, então, é uma simples questão de calcular qual bloco do mapa de bits contém o bit para o nó-i sendo liberado e a localizá-lo, utilizando o mecanismo normal de *cache*. Uma vez que o bloco foi localizado, o bit correspondente ao nó-i liberado é configurado como 0. As zonas são liberadas do mapa de bits de zonas da mesma maneira.

Logicamente, quando um arquivo está para ser criado, o sistema de arquivos deve pesquisar o primeiro nó-i livre pelos blocos do mapa de bits um por vez. Esse nó-i, então, é alocado para o novo arquivo. De fato, a cópia na memória do superbloco tem um campo que aponta para o primeiro nó-i livre; então, nenhuma pesquisa é necessária até depois que um nó for utilizado, quando o ponteiro deve ser atualizado para apontar para o novo próximo nó-i livre, o qual freqüentemente será o seguinte ou um próximo. De maneira semelhante, quando um nó-i é liberado, uma verificação é feita para ver se o nó-i livre vem antes do atualmente apontado; e o ponteiro é atualizado se necessário. Se cada entrada de nó-i no disco estiver cheia, a rotina de pesquisa retornará um 0, que é a razão por que o nó-i 0 não é utilizado (i. e., para que ele possa ser utilizado para indicar a falha na pesquisa). (Quando *mkfs* cria um novo sistema de arquivos, ele zera o nó-i 0 e configura o bit mais baixo no mapa de bits como 1, de modo que o sistema de arquivos nunca tentará alocá-lo.) Tudo o que foi dito aqui sobre os mapas de bits de nós-i também se aplica ao mapa de bits de zonas; logicamente ele é pesquisado quanto à primeira zona livre quando espaço é necessário, mas um ponteiro para a primeira zona livre é mantido para eliminar a maior parte da necessidade de pesquisas seqüenciais pelo mapa de bits.

Com essa fundamentação, agora podemos explicar a diferença entre zonas e blocos. A idéia por trás das zonas é ajudar a assegurar que blocos de disco pertencentes ao mesmo arquivo estejam localizados no mesmo cilindro, melhorando o desempenho quando o arquivo é lido seqüencialmente. A abordagem escolhida é tornar possível alocar diversos blocos por vez. Se, por exemplo, o tamanho de um bloco for de 1K e o tamanho de zona for de 4K, o mapa de bits de zonas irá monitorar zonas, não blocos. Um disco de 20M tem 5K zonas de 4K, daí 5K bits em seu mapa de zonas.

A maior parte do sistema de arquivos trabalha com blocos. As transferências de disco são sempre um bloco por vez, e o *cache* também trabalha com blocos individuais. Somente algumas partes do sistema que monitoram endereços físicos de disco (p. ex., o mapa de bits de zonas e os nós-i) têm conhecimento das zonas.

Algumas decisões de projeto tiveram de ser feitas ao desenvolver-se o sistema de arquivos do MINIX. Em 1985, quando o MINIX foi concebido, as capacidades de disco eram pequenas, e acreditava-se que a maioria dos usuários teria somente disquetes. Uma decisão foi tomada para restringir endereços de disco para 16 bits no sistema de arquivos V1, principalmente para ser capaz de armazenar muitos deles nos blocos indiretos. Com um número de zona de 16 bits e uma zona de 1K, apenas 64K de zonas podem ser endereçadas, limitando os discos a 64 M. Isso era um espaço enorme de armazenamento naquela época e pensou-se que à medida que os discos aumentassem, seria fácil alternar para zonas de 2K ou de 4K, sem alterar o tamanho do bloco. Os números de zona de 16 bits também tornaram fácil manter o tamanho dos nós-i em 32 bytes.

À medida que o MINIX desenvolvia-se, e os discos maiores tornavam-se cada vez mais comuns, tornou-se óbvio que alterações eram desejáveis. Muitos arquivos são menores que 1K, então, aumentar o tamanho de bloco significaria desperdiçar largura de banda de disco, lendo e gravando principalmente blocos vazios e desperdiçando preciosa memória principal, armazenando-os no *cache*. O ta-

manho da zona poderia ter sido aumentado, mas um tamanho de zona maior significa mais espaço em disco desperdiçado e era ainda desejável manter uma operação eficiente em discos pequenos. Outra alternativa razoável seria ter tamanhos diferentes de zona de acordo com o tamanho dos dispositivos.

No fim, decidiu-se aumentar o tamanho dos ponteiros de disco para 32 bits. Isso torna possível o sistema de arquivos MINIX V2 lidar com tamanhos de dispositivo de até 4 terabytes com blocos e zonas de 1K. Em parte, essa decisão baseou-se em outras decisões sobre o que deve estar no nó-i, o que tornou razoável aumentar o tamanho do nó-i para 64 bytes.

As zonas também introduzem um problema inesperado, melhor ilustrado por um exemplo simples, novamente com zonas de 4K e blocos de 1K. Suponha que um arquivo tenha 1K de comprimento, o que significa que 1 zona foi alocada para ele. Os blocos entre 1K e 4K contêm lixo (resíduo do proprietário anterior), mas nenhum dano é causado porque o tamanho de arquivo claramente é marcado no nó-i como 1K. De fato, os blocos contendo lixo não serão lidos no *cache* de blocos, uma vez que as leituras são feitas por blocos, não por zonas. As leituras além do fim de um arquivo sempre retornam uma contagem de 0 e nenhum dado.

Agora alguém move para 32768 e grava 1 byte. O tamanho do arquivo agora é alterado para 32769. Movimentos subseqüentes para 1K seguidos por tentativas de ler os dados agora serão capazes de ler o conteúdo anterior do bloco, uma importante falha na segurança.

A solução é verificar essa situação quando uma gravação é feita além do fim de um arquivo e explicitamente zerar todos os blocos que ainda não foram alocados na zona que era antes a última. Embora essa situação raramente ocorra, o código precisa lidar com ela, o que torna o sistema ligeiramente mais complexo.

5.6.4 Nós-i

O leiaute do nó-i do MINIX é dado na Figura 5-30. Ele é quase o mesmo de um nó-i padrão do UNIX. Os ponteiros de zona de disco são de 32 bits e há somente 9 ponteiros, 7 diretos e 2 indiretos. Os nós-i do MINIX ocupam 64 bytes, o mesmo que os nós-i padrão do UNIX e há espaço disponível para um 10° (triplo indireto) ponteiro, embora sua utilização não seja suportada pela versão padrão do sistema de arquivos. Os tempos de acesso, de modificação e de alteração do nó-i no MINIX são padrão, como no UNIX. O último destes é atualizado para quase cada operação de arquivo exceto para uma leitura do arquivo.

Quando um arquivo é aberto, seu nó-i é localizado e carregado na tabela *inode* na memória, onde permanece até que o arquivo seja fechado. A tabela *inode* tem alguns campos adicionais não-presentes no disco, como o número e o dispositivo do nó-i; então, o sistema de arquivos sabe onde regravar se ele for modificado enquanto na memória. Ela também tem um contador por nó-i. Se o mesmo arquivo for aberto mais de uma vez, somente uma cópia do nó-i é mantida na memória, mas o contador é incrementado cada vez que o arquivo é aberto, e decrementado cada vez que o arquivo é fechado. Somente quando o contador por fim alcança zero é que o nó-i é removido da tabela. Se foi modificado desde que foi carregado na memória, ele também é regravado no disco.

A função principal de um nó-i de arquivo é informar onde os blocos de dados estão. Os primeiros sete números de zona são dados diretamente no próprio nó-i. Para a distribuição padrão, com zonas e blocos de 1K, arquivos até 7K não necessitam de blocos indiretos. Além de 7K, zonas indiretas são necessárias, utilizando o esquema da Figura 5-10, exceto que somente blocos simples e indiretos duplos são utilizados. Com blocos e zonas de 1K e números de zona de 32 bits, um bloco indireto simples armazena 256 entradas, o que representa um quarto de megabyte de armazenamento. O bloco indireto duplo aponta para 256 blocos indiretos simples, dando acesso a até 64 megabytes. O tamanho máximo de um sistema de arquivos MINIX é de 1 G, portanto uma modificação para utilizar o bloco indireto triplo ou tamanhos maiores de zona poderia ser útil se fosse desejável acessar arquivos muito grandes em um sistema MINIX.

O nó-i também armazena as informações de modo, que dizem qual é o tipo de um arquivo (comum, diretório, de bloco especial, de caractere especial, canalização) e dá os bits de proteção, SETUID e SETGID. O campo número de vínculos no nó-i registra quantas entradas de diretório apontam para o nó-i, então, o sistema de arquivos sabe quando liberar o armazenamento do arquivo. Esse campo não deve ser confundido com o contador (presente somente na tabela *inode* na memória, não no disco) que diz quantas vezes o arquivo está atualmente aberto, geralmente por processos diferentes.

5.6.5 *Cache* de Blocos

O MINIX utiliza um *cache* de blocos para melhorar o desempenho do sistema de arquivos. O *cache* é implementado como uma matriz de buffers, cada um consistindo em um cabeçalho contendo ponteiros, contadores e sinalizadores, e um corpo com lugar para um bloco de disco. Todos os buffers que não estão em utilização são encadeados juntos em uma lista duplamente encadeada, do mais recentemente utilizado (MRU) para o menos recentemente utilizado (LRU), como ilustrado na Figura 5-31.

Além disso, para ser capaz de rapidamente determinar se um dado bloco está no *cache* ou não, uma tabela de *hash* é utilizada. Todos os buffers contendo um bloco que tem código de *hash* *k* estão encadeados juntos em uma lista encadeada simples apontada pela entrada *k* na tabela de *hash*. A função de *hash* simplesmente extrai os *n* bits de ordem inferior do número do bloco, portanto blocos de dispositivos diferentes aparecem na mesma cadeia de *hash*. Cada buffer está em uma dessas cadeias. Quando o sistema de arquivos é inicializado depois que o MINIX é inicializa-

16 bits

64 bytes

Campo	Descrição
Modo	Tipo do arquivo e bits rwx
Número de vínculos	Entradas de diretório para esse arquivo
Uid	Identifica usuário que possui o arquivo
Gid	Grupo do proprietário
Tamanho do arquivo	Número de bytes no arquivo
Tempo de acesso	Todos os tempos são em segundos desde 1º de Janeiro de 1970
Tempo de modificação	
Tempo de alteração de status	
Zona 0	Números de zona para as primeiras sete zonas de dados no arquivo
Zona 1	
Zona 2	
Zona 3	
Zona 4	
Zona 5	
Zona 6	
Zona indireta	Utilizado para arquivos maiores que 7 zonas
Zona indireta dupla	
Não-utilizada	(Poderia ser utilizado para zona indireta tripla)

Figura 5-30 O nó-i do MINIX.

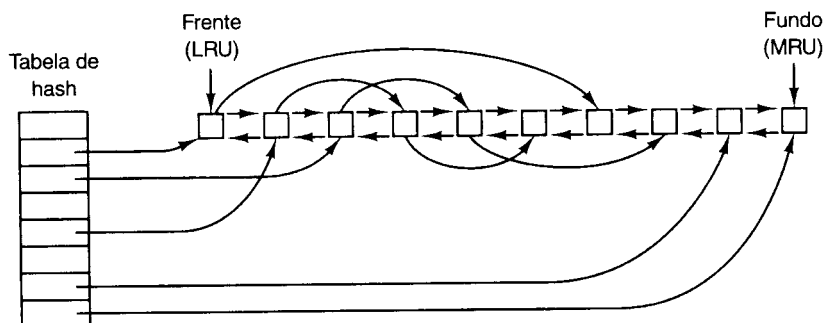



Figura 5-31 As listas encadeadas utilizadas pelo *cache* de blocos.

do, todos os buffers não estão em uso, naturalmente, e todos estão em uma única cadeia apontada pela entrada 0 da tabela de *hash*. Nesse momento todas as outras entradas da tabela de *hash* contêm um ponteiro nulo, mas uma vez que o sistema inicia, os buffers serão removidos da cadeia 0 e outras cadeias serão construídas.

Quando o sistema de arquivos necessita de um bloco, ele chama um procedimento, *get_block*, que computa o código de *hash* para esse bloco e pesquisa a lista apropriada. *Get_block* é chamado com um número de dispositivo assim como um número de bloco, e a pesquisa compara ambos os números com os campos correspondentes na cadeia de buffers. Se um buffer que contém o bloco é localizado, um contador no cabeçalho do buffer é incrementado para mostrar que o bloco está em uso, e um ponteiro para ele é retornado. Se um bloco não é localizado na lista de *hash*, o primeiro buffer na lista de LRU pode ser utilizado; não é garantido estar ainda em utilização, e o bloco que contém pode ser expulso para liberar o buffer.

Uma vez que um bloco foi escolhido para expulsão, outro sinalizador em seu cabeçalho é verificado para ver se o bloco foi modificado desde que foi lido. Se tiver sido, ele é regravado no disco. Nesse ponto, o bloco necessário é lido mediante o envio de uma mensagem à tarefa de disco. O sistema de arquivos é suspenso até que o bloco chegue, momento em que ele continua, e um ponteiro para o bloco é retornado para o chamador.

Quando o procedimento que solicitou o bloco completar seu trabalho, chama outro procedimento, *put_block*, para liberar o bloco. Normalmente, um bloco será utilizado imediatamente e, então, liberado, mas uma vez que é possível que solicitações adicionais para um bloco sejam feitas antes de ele ser liberado, *put_block* decrementa o contador de uso e coloca o buffer de volta na lista de LRU somente quando o contador de uso voltar a zero. Enquanto o contador não for zero, o bloco permanece no limbo.

Um dos parâmetros para *put_block* informa qual classe de bloco (p. ex., nós-i, diretório, dados) está sendo liberada. Dependendo da classe, duas decisões-chaves são feitas:

1. Colocar o bloco na frente ou no fundo da lista de LRU.
2. Gravar o bloco (se modificado) para disco, imediatamente ou não.

Os blocos que podem não ser necessários novamente em seguida, como superblocos, vão na frente da lista para que sejam reivindicados da próxima vez que um buffer livre for necessário. Todos os outros blocos vão no fundo da lista no verdadeiro estilo LRU.

Um bloco modificado não é regravado até que qualquer um destes dois eventos ocorra:

1. Alcançar a frente da cadeia de LRU e ser expulso.
2. Uma chamada de sistema SYNC for executada.

SYNC não varre a cadeia de LRU, mas em vez disso percorre a matriz de buffers no *cache*. Mesmo se um buffer ainda não foi liberado, se foi modificado, SYNC irá localizá-lo e assegurará que a cópia em disco seja atualizada.

Há uma exceção, contudo. Um superbloco modificado é gravado em disco imediatamente. Em uma versão mais antiga do MINIX, um superbloco era modificado quando um sistema de arquivos era montado, e o propósito da gravação imediata era reduzir a chance de corromper o sistema de arquivos em caso de uma queda. Os superblocos não são modificados agora, então, o código para gravá-los imediatamente é um anacronismo. Na configuração-padrão, nenhum outro bloco é gravado imediatamente. Entretanto, modificando-se a definição padrão de *ROBUST* no arquivo de configuração de sistema, *include/minix/config.h*, o sistema de arquivos pode ser compilado para marcar blocos de nós-i, de diretórios, de mapa de bits ou indiretos de tal modo que eles sejam gravados imediatamente na liberação. Isso com objetivo de fazer o sistema de arquivos mais robusto; o preço a ser pago é a operação mais lenta. Se isso será efetivo, não é claro. Uma falta de energia que ocorra quando todos os blocos ainda não foram gravados causará uma dor de cabeça se um nó-i ou um bloco de dados for perdido.

Note que o sinalizador de cabeçalho indicando que um bloco foi modificado é configurado pelo procedimento dentro do sistema de arquivos que solicitou e utilizou o bloco. Os procedimentos *get_block* e *put_block* estão preocupados apenas com manipuladores das listas encadeadas. Eles não têm nenhuma idéia de qual procedimento do sistema de arquivos quer qual bloco ou por quê.

5.6.6 Diretórios e Caminhos

Outro subsistema importante dentro do sistema de arquivos é o gerenciamento de diretórios e os nomes de caminho. Muitas chamadas de sistema, como OPEN, têm um nome de arquivo como parâmetro. O que é realmente necessário é o nó-i para esse arquivo, então, cabe ao sistema de arquivos localizar o arquivo na árvore de diretórios e obter seu nó-i.

Um diretório MINIX consiste em um arquivo contendo entradas de 16 bytes. Os primeiros 2 bytes formam um número de nó-i de 16 bits e os 14 bytes restantes são o nome do arquivo. Essa é a mesma entrada de diretório tradicional do UNIX que vimos na ilustração da Figura 5-13. Para procurar o caminho */usr/ast/mbox*, o sistema primeiro localiza *usr* no diretório-raiz, então, localiza *ast* em *usr* e, por fim, localiza *mbox* em */usr/ast*. A pesquisa real segue um componente do caminho por vez como ilustrado na Figura 5-14.

A única complicação é o que acontece quando um sistema de arquivos montado é encontrado. A configuração normal para o MINIX e para muitos outros sistemas tipo UNIX é ter um pequeno sistema de arquivos-raiz contendo os arquivos necessários para iniciar o sistema e fazer a manutenção básica de sistema, e ter a maioria dos arquivos, incluindo diretórios dos usuários, em um dispositivo separado montado em */usr*. Esse é um bom momento para ver

como a montagem é feita. Quando o usuário digita o comando

```
mount /dev/hd2c /usr
```

no terminal, o sistema de arquivos contido na partição 2 do disco rígido é montado sobre */usr* no sistema de arquivos raiz. Os sistemas de arquivos antes e depois da montagem são mostrados na Figura 5-32.

A chave para todo o negócio da montagem é um sinalizador configurado na cópia em memória do nó-i de */usr* após uma montagem bem-sucedida. Esse sinalizador indica que o nó-i foi montado. A chamada MOUNT também carrega o superbloco do sistema de arquivos recentemente montado na tabela *super_block* e configura dois ponteiros nela. Além disso, ele coloca o nó-i raiz do sistema de arquivos montado na tabela *inode*.

Na Figura 5-29, vemos que os superblocos na memória contêm dois campos relacionados com sistemas de arquivos montados. O primeiro desses, o *nó-i-do-sistema-de-arquivo-montado*, é configurado para apontar para o nó-i raiz do sistema de arquivos recentemente montado. O segundo, *nó-i-montado*, é configurado para apontar para o nó-i onde ocorreu a montagem, nesse caso o nó-i de */usr*. Esses dois ponteiros servem para conectar o sistema de arquivos montado à raiz e representam a "cola" que une o sistema de arquivos montado à raiz [mostrado como os pontos na Figura 5-32(c)]. Essa cola é o que faz os sistemas de arquivos montados funcionarem.

Quando um caminho como */usr/ast/f2* está sendo pesquisado, o sistema de arquivos olha em um sinalizador no nó-i de */usr* e sabe que deve continuar a pesquisar no nó-i raiz do sistema de arquivos montado em */usr*. A pergunta é: "Como localiza esse nó-i raiz"?

A resposta é simples e direta. O sistema pesquisa todos os superblocos na memória até encontrar aquele cujo campo *nó-i montado* aponta para */usr*. Esse deve ser o superbloco para o sistema de arquivos montado em */usr*. Uma vez que ele tem o superbloco, é fácil seguir o outro ponteiro para localizar o nó-i raiz para o sistema de arquivos montado. Agora, o sistema de arquivos pode continuar a pesquisar. Nesse exemplo, ele procura *ast* no diretório-raiz da partição do disco rígido 2.

5.6.7 Descritores de Arquivos

Uma vez que um arquivo foi aberto, um descritor de arquivo é retornado para o processo de usuário para utilização em subseqüentes chamadas READ e WRITE. Nesta seção, veremos como descritores de arquivos são gerenciados dentro do sistema de arquivos.

Como o *kernel* e com o gerenciador de memória, o sistema de arquivos mantém parte da tabela de processos dentro de seu espaço de endereços. Três dos seus campos são de interesse particular. Os primeiros dois são ponteiros para os nós-i para o diretório-raiz e para o diretório de trabalho. As pesquisas de caminho, como a da Figura 5-14, sempre começam em um ou em outro, dependendo de o caminho ser absoluto ou relativo. Esses ponteiros são alterados pelas chamadas de sistema CHROOT e CHDIR para apontar para a nova raiz ou para novo diretório de trabalho respectivamente.

O terceiro campo interessante na tabela de processos é uma matriz indexada pelo número do descritor de arqui-

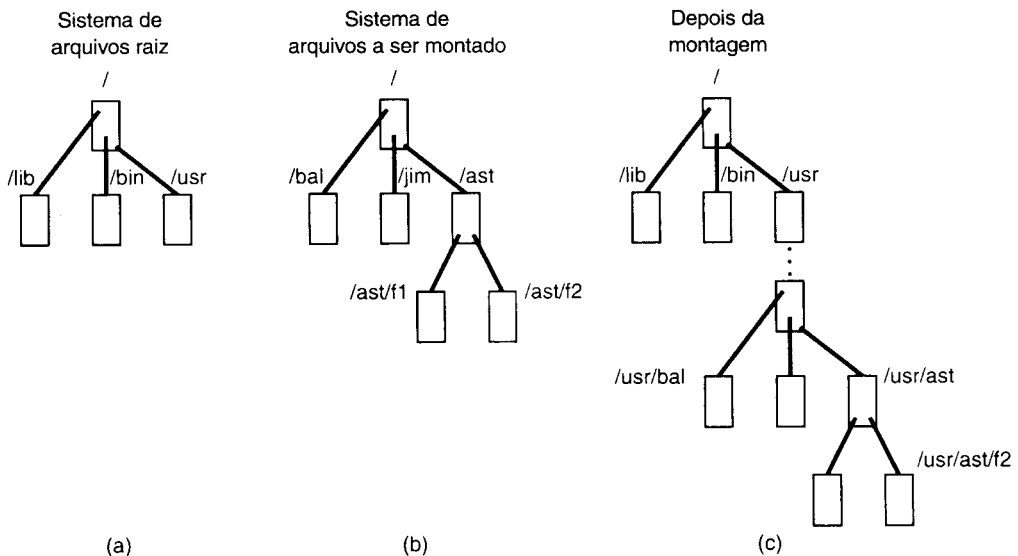



Figura 5-32 (a) Sistema de arquivos raiz. (b) Um sistema de arquivos não-montado. (c) O resultado da montagem do sistema de arquivos de (b) em */usr*.

vo. Ela é utilizada para localizar o arquivo adequado quando um descritor de arquivo é apresentado. À primeira vista, talvez pareçam suficientes ter a *k-ésima* entrada nessa matriz apontando simplesmente para o nó-i do arquivo pertencente ao descritor de arquivo *k*. Afinal de contas, o nó-i é buscado na memória quando o arquivo está aberto e é mantido aí até que seja fechado, assim ele fica seguro de estar disponível.

Infelizmente, esse plano simples falha porque os arquivos podem ser compartilhados de maneiras sutis no MINIX (assim como no UNIX). O problema surge porque associado com cada arquivo está um número de 32 bits que indica o próximo byte a ser lido ou gravado. É esse número, chamado **posição do arquivo**, que é alterado pelo chamada de sistema LSEEK. O problema pode ser facilmente declarado: "Onde o ponteiro de arquivo deve estar armazenado? "

A primeira possibilidade é colocá-lo no nó-i. Infelizmente, se dois ou mais processos tiverem aberto ao mesmo tempo o mesmo arquivo, todos eles deverão ter os próprios ponteiros de arquivo, uma vez que seria péssimo que uma LSEEK de um processo afetasse a próxima leitura de um processo diferente. A conclusão: a posição do arquivo não pode entrar no nó-i.

Que tal colocá-lo na tabela de processos? Por que não ter uma segunda matriz, paralela à matriz de descritores de arquivos, dando a posição atual de cada arquivo? Essa idéia também não funciona, mas a razão é mais sutil. Basicamente, o problema vem da semântica da chamada de sistema FORK. Quando um processo bifurca, exige-se que pai e filho compartilhem um único ponteiro que dá a posição atual de cada arquivo aberto.

Para entender melhor o problema, considere o caso de um *script* do *shell* cuja saída foi redirecionada para um arquivo. Quando o *shell* bifurca o primeiro programa, sua posição de arquivo para saída-padrão é 0. Essa posição, então, é herdada pelo filho, que grava, digamos, 1K de saída. Quando o filho termina, a posição de arquivo compartilhada agora deve ser de 1K.

Agora o *shell* lê algo mais do *script* de *shell* e cria outro filho. É essencial que a segundo filho herde uma posição de arquivo em um 1K do *shell*, de modo que ele começará a gravar no lugar que o primeiro programa parou. Se o *shell* não compartilhou a posição de arquivo com seus filhos, o segundo programa sobrescreveria a saída do primeiro, em vez de anexar a ela.

Como resultado, não é possível colocar a posição de arquivo na tabela de processos. Ela realmente deve ser compartilhada. A solução utilizada no MINIX é introduzir uma nova tabela compartilhada, *filp*, que contém todas as posições de arquivo. Sua utilização é ilustrada na Figura 5-33. Por ter a posição de arquivo verdadeiramente compartilhada, a semântica de FORK pode ser implementada corretamente, e *scripts* de *shell* funcionam adequadamente.

Embora a única coisa que a tabela *filp* realmente deve conter é a posição de arquivo compartilhado, é conveniente colocar o ponteiro de nó-i aí, também. Assim, tudo que a matriz de descritores de arquivos na tabela de processos contém é um ponteiro para uma entrada *filp*. A entrada *filp* também contém o modo do arquivo (bits de permissão), alguns sinalizadores que indicam se o arquivo foi aberto em um modo especial e uma contagem do número de processos que o estão utilizando; então, o sistema de arquivos pode dizer quando terminou o último processo que está utilizando a entrada, a fim de reivindicá-la.

5.6.8 Bloqueio de Arquivos

Há ainda outro aspecto de gerenciamento do sistema de arquivos que requer uma tabela especial. Isso é o bloqueio de arquivos. O MINIX suporta o mecanismo de comunicação interprocesso do POSIX de **bloqueio de arquivo consultivo**. Este último permite que qualquer parte, ou múltiplas partes, de um arquivo sejam marcadas como bloqueadas. O sistema operacional não impõe bloquear, mas espera-se que os processos sejam mais comportados e pesquisem bloqueios em um arquivo antes de fazer qualquer coisa que geraria conflitos com outro processo.

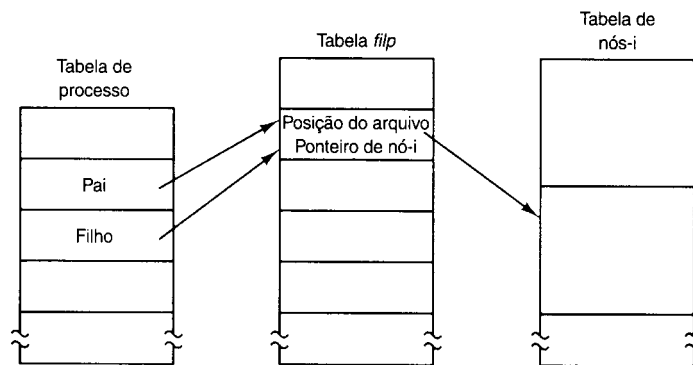



Figura 5-33 Como posições de arquivo são compartilhadas entre um pai e um filho.

As razões para oferecer uma tabela separada para bloqueios são semelhantes às justificativas para a tabela *filp*, discutida na seção anterior. Um único processo pode ter mais de um bloqueio ativo, e diferentes partes de um arquivo podem ser bloqueadas por mais de um processo (embora, naturalmente, os bloqueios não possam sobrepor-se), assim nem a tabela de processos nem a tabela *filp* são um bom lugar para registrar bloqueios. Uma vez que um arquivo pode ter mais de um bloqueio colocado sobre ele, o nó-i também não é um bom lugar.

O MINIX utiliza outra tabela, a tabela *file_lock*, para registrar todos os bloqueios. Cada entrada nessa tabela tem espaço para um tipo de bloqueio, indicando se o arquivo é bloqueado para leitura ou para escrita, o ID do processo que está fazendo o bloqueio, um ponteiro para o nó-i do arquivo bloqueado e os deslocamentos do primeiro e do último bytes da região bloqueada.

5.6.9 Canalizações e Arquivos Especiais

As canalizações (*pipes*) e arquivos especiais diferem de arquivos comuns de uma maneira importante. Quando um processo tenta ler ou gravar em um arquivo de disco, é certo que a operação irá completar-se dentro de algumas centenas de milissegundos no máximo. No pior caso, dois ou três acessos de disco talvez sejam necessários, não mais. Ao ler de uma canalização, a situação é diferente: se a canalização estiver vazia, o leitor precisará esperar até que algum outro processo coloque dados na canalização, o que talvez leve horas. De maneira semelhante, ao ler de um terminal, um processo terá de esperar até que alguém digite algo.

Como conseqüência, a regra normal do sistema de arquivos de tratar uma solicitação até que ela termine o trabalho não funciona. É necessário suspender essas solicitações e reiniciá-las mais tarde. Quando um processo tenta ler ou gravar de uma canalização, o sistema de arquivos pode verificar o estado da canalização imediatamente para ver se a operação pode ser completada. Se puder, ela é completada, mas, se não puder, o sistema de arquivos registra os parâmetros da chamada de sistema na tabela de processos, assim ele pode reiniciar o processo quando a hora chegar.

Note que o sistema de arquivos não precisa realizar qualquer ação para ter o chamador suspenso. Tudo que ele precisa fazer é deixar de enviar uma resposta, deixando o chamador esperar a resposta bloqueado. Assim, depois de suspender um processo, o sistema de arquivos volta ao seu laço principal, esperando a próxima chamada de sistema. Logo que outro processo modifica o estado da canalização, de modo que o processo suspenso possa continuar, o sistema de arquivos ativa um sinalizador para que da próxima vez que passar pelo laço principal ele extraia os parâmetros do processo suspenso da tabela de processos e execute a chamada.

A situação com terminais e com outros arquivos especiais de caractere é ligeiramente diferente. O nó-i para cada arquivo especial contém dois números, o dispositivo principal e o dispositivo secundário. O número de dispositivo principal indica a classe do dispositivo (p. ex., disco de RAM, disquete, disco rígido, terminal). Ele é utilizado como um índice em uma tabela do sistema de arquivos que o mapeia para o número da tarefa correspondente (i. e., *driver* de E/S). De fato, o dispositivo principal determina qual *driver* de E/S chamar. O número do dispositivo secundário é passado para o *driver* como um parâmetro. Ele especifica qual dispositivo deve ser utilizado, por exemplo, terminal 2 ou unidade 1.

Em alguns casos, principalmente dispositivos de terminal, o número de dispositivo secundário codifica algumas informações sobre uma categoria de dispositivos tratada por uma tarefa. Por exemplo, o console primário do MINIX, */dev/console*, é o dispositivo 4, 0 (primário, secundário). Consoles virtuais são tratados pela mesma parte do software de *driver*. Estes últimos são os dispositivos */dev/ttyc1* (4,1), */dev/ ttyc2* (4, 2) e assim por diante. Terminais de linha serial necessitam de software de baixo nível diferente, e a esses dispositivos, */dev/tty00* e */dev/tty01*, são atribuídos os números de dispositivo 4, 16 e 4, 17. De maneira semelhante, terminais de rede utilizam *drivers* de *pseudo-terminal* e estes últimos também necessitam de software de baixo nível diferente. No MINIX, a esses dispositivos, *ttyp0*, *ttyp1* etc., são atribuídos números de dispositivo como 4, 128 e 4, 129. Cada um desses *pseudodispositivos* tem um dispositivo associado, *ptyp0*, *ptyp1*, etc. Os pares primário, secundário de número de dispositivo para esses são 4,192, 4,193 e assim por diante. Esses números são escolhidos para tornar fácil para a tarefa de *driver* chamar as funções de baixo nível requeridas para cada grupo de dispositivos. Não há nenhuma expectativa de que qualquer pessoa irá equipar um sistema MINIX com 192 ou com mais terminais.

Quando um processo lê de um arquivo especial, o sistema de arquivos extrai os números de dispositivo primário e secundário do nó-i do arquivo e utiliza o número de dispositivo principal como um índice em uma tabela de sistema de arquivos a fim de mapeá-lo para o número de tarefa correspondente. Uma vez que tenha o número de tarefa, o sistema de arquivos envia à tarefa uma mensagem, incluindo como parâmetros o dispositivo secundário, a operação a ser executada, o número de processo e o endereço de buffer do chamador e o número de bytes a serem transferidos. O formato é o mesmo que da Figura 3-15, exceto que *POSITION* não é utilizado.

Se o *driver* for capaz de executar o trabalho imediatamente (p. ex., uma linha de entrada já foi digitada no terminal), ele copia os dados dos próprios buffers internos para o usuário e envia ao sistema de arquivos uma mensagem de resposta, dizendo que o trabalho está feito. O sistema de arquivos, então, envia uma mensagem de resposta para o usuário, e a chamada termina. Note que o *driver* não copia os dados para o sistema de arquivos. Os dados provenientes dos dispositivos de blocos vão pelo *cache* de blocos, mas dados de arquivos especiais de caractere não.

Por outro lado, se o *driver* não for capaz de executar o trabalho, ele registra os parâmetros da mensagem em suas

tabelas internas e imediatamente envia uma resposta para o sistema de arquivos, dizendo que a chamada não pôde ser completada. Nesse ponto, o sistema de arquivos está na mesma situação que estaria se tivesse descoberto que alguém está tentando ler de uma canalização vazia. Ele registra o fato de que o processo está suspenso e espera a próxima mensagem.

Quando o *driver* adquiriu dados suficientes para completar a chamada, transfere-os ao buffer do usuário ainda bloqueado e, então, envia ao sistema de arquivos uma mensagem que informa o que fez. Tudo o que o sistema de arquivos precisa fazer é enviar uma mensagem de resposta ao usuário para desbloqueá-lo e informar o número de bytes transferidos.

5.6.10 Um Exemplo: a Chamada de Sistema READ

Como veremos brevemente, a maior parte do código do sistema de arquivos é dedicada para dar conta de chamadas de sistema. Portanto, é apropriado concluirmos essa visão geral com um breve esboço de como a chamada mais importante, READ, funciona.

Quando um programa de usuário executa a declaração

```
n = read(fd, buffer, nbytes);
```

para ler um arquivo comum, o procedimento de biblioteca *read* é chamado com três parâmetros. Ele constrói uma mensagem que contém tais parâmetros, junto com o código para READ como o tipo de mensagem, envia a mensagem para o sistema de arquivos e fica suspenso aguardando a resposta. Quando a mensagem chega, o sistema de arquivos utiliza o tipo de mensagem como um índice em suas tabelas para chamar o procedimento que trata a leitura.

Esse procedimento extrai o descritor de arquivo da mensagem e utiliza-o para localizar a entrada *filp*; então, o nó-i para o arquivo a ser lido (veja a Figura 5-33). A solicitação, então, é dividida em pedaços de tal modo que cada pedaço ajusta-se dentro de um bloco. Por exemplo, se a posição do arquivo atual for 600 e 1K bytes tiverem sidos solicitados, a solicitação será dividida em duas partes, de 600 até 1023 e de 1024 até 1623 (supondo blocos de 1K).

Para cada um desses pedaços, por sua vez, é verificado se o bloco relevante está no *cache*. Se o bloco não estiver presente, o sistema de arquivos seleciona o buffer menos recentemente utilizado que não esteja em uso e reivindica-o, enviando uma mensagem à tarefa de disco para regravá-lo se ele estiver sujo. Então, é solicitado à tarefa de disco que busque o bloco a ser lido.

Uma vez que o bloco está no *cache*, o sistema de arquivos envia uma mensagem à tarefa de sistema, pedindo-lhe para copiar os dados para o lugar apropriado no buffer do usuário (i. e., bytes 600 até 1023 para o início do buffer e bytes 1024 até 1623 para o deslocamento 424 dentro do buffer). Depois que a cópia foi feita, o sistema de arquivos envia uma mensagem de resposta para o usuário especificando quantos bytes foram copiados.

Quando a resposta volta para o usuário, a função de biblioteca *read* extrai o código de resposta e retorna-o como o valor da função para o chamador

Há um passo extra que não é realmente parte da chamada READ em si. Depois que o sistema de arquivos completa uma leitura e envia uma resposta, ele, então, inicia uma leitura do próximo bloco, desde que a leitura seja de um dispositivo de bloco e outras condições particulares sejam satisfeitas. Uma vez que leituras seqüenciais de arquivo são comuns, é razoável esperar que o próximo bloco em um arquivo seja solicitado na próxima solicitação de leitura e isso torna possível que o bloco desejado já esteja no *cache* quando for necessário.

5.7 IMPLEMENTAÇÃO DO SISTEMA DE ARQUIVOS DO MINIX

O sistema de arquivos do MINIX é relativamente grande (mais de 100 páginas em C), mas bem simples e direto. As solicitações para executar chamadas de sistema chegam, são executadas e respostas são enviadas. Nas seções a seguir, nós o estudaremos um arquivo por vez, indicando os destaques. O próprio código contém muitos comentários que ajudam o leitor.

Ao examinar o código de outras partes do MINIX geralmente olhamos primeiro o laço principal de um processo e, então, olhamos as rotinas que tratam os diferentes tipos de mensagem. Organizaremos nossa abordagem para o sistema de arquivos de maneira diferente. Primeiro, passaremos pelos subsistemas importantes (gerenciamento de *cache*, gerenciamento de nós-i, etc.). Então, veremos o laço principal e as chamadas de sistema que operam sobre arquivos. Em seguida, veremos chamadas de sistema que operam sobre diretórios. Por fim, discutiremos as chamadas de sistema restantes que não entram em nenhuma categoria.

5.7.1 Arquivos Cabeçalho e Estruturas de Dados Globais

Como o *kernel* e o gerenciador de memória, várias estruturas de dados e tabelas utilizadas no sistema de arquivos são definidas em arquivos de cabeçalho. Algumas dessas estruturas de dados são colocadas em arquivos de cabeçalho de sistema em *include/* e seus subdiretórios. Por exemplo, *include/sys/stat.h* define o formato por meio do qual chamadas de sistema podem oferecer as informações de nó-i para outros programas, e a estrutura de uma entrada de diretório é definida em *include/sys/dir.h*. Ambos arquivos são exigidos pelo POSIX. O sistema de arquivos é afetado por diversas definições contidas no arquivo global de configuração *include/minix/config.h*, como a macro *ROBUST* que define se estruturas de dados importantes do sistema de arquivos sempre serão gravadas imediatamente

no disco e *NR_BUFS* e *NR_BUF_HASH*, que controlam o tamanho do *cache* de blocos.

Cabeçalhos do Sistema de Arquivos

Os arquivos de cabeçalho do próprio sistema de arquivos estão no diretório de fontes do sistema de arquivos *src/fs/*. Muitos nomes de arquivo serão familiares a partir do estudo de outras partes do sistema MINIX. O arquivo de cabeçalho-mestre do sistema de arquivos, *fs.h* (linha 19400), é muito semelhante a *src/kernel/kernel.h* e *src/mm/mm.h*. Ele inclui outros arquivos de cabeçalho necessários por todos os arquivos-fonte de C no sistema de arquivos. Como nas outras partes do MINIX, o cabeçalho-mestre do sistema de arquivos inclui *const.h*, *type.h*, *proto.h* e *glo.h* próprios do sistema de arquivos. Veremos estes últimos a seguir.

Const.h (Linha 19500) define algumas constantes como tamanhos de tabela e sinalizadores, que são utilizadas por todo o sistema de arquivos. O MINIX já tem uma história. Uma versão anterior teve um sistema de arquivos diferente, e para usuários que queiram acessar arquivos gravados pela versão anterior, é oferecido suporte para ambos, o sistema de arquivos *V1* antigo e o *V2* atual. O superbloco de um sistema de arquivos contém um **número mágico** para que o sistema operacional possa identificar o tipo; as constantes *SUPER_MAGIC* e *SUPER_V2* definem esses números. O suporte de versões antigas não é algo que se possa ler em textos teóricos, mas é sempre um interesse para o implementorador de uma nova versão de qualquer software. Deve-se decidir quanto esforço será dedicado para tornar a vida mais fácil para o usuário da versão antiga. Veremos vários lugares no sistema de arquivos onde o suporte para a versão antiga é uma questão importante.

Type.h (Linha 19600) define as estruturas de nó-i V1 (antigas) e V2 (novas) como são colocadas no disco. O nó-i V2 é duas vezes maior que o antigo, que foi projetado para ser compacto em sistemas sem disco rígido e disquetes de 360KB. A nova versão oferece espaço para os três campos de tempo que os sistemas UNIX oferecem. No nó-i V1, havia somente um campo de tempo, mas um STAT ou FSTAT "fingia" e retornava uma estrutura *stat* contendo todos os três campos. Aqui há uma pequena dificuldade em oferecer suporte para as duas versões do sistema de arquivos. Isso é indicado pelo comentário na linha 19616. O software MINIX mais antigo espera que o tipo *gid_t* tenha um tamanho de 8 bits, então, *d2_gid* deve ser declarado como tipo *u16_t*.

Proto.h (linha 19700) oferece protótipos de função em formas aceitáveis tanto para compiladores antigos, K&R como para compiladores padrão ANSI C mais recentes. É um arquivo longo, mas não de grande interesse. Entretanto, há um ponto a notar: como há tantas chamadas de sistema diferentes tratadas pelo sistema de arquivos, e por causa da maneira como o sistema de arquivos é organizado, as várias funções *do_xxx* estão dispersas por vários arquivos. *Proto.h* é organizado por arquivo e é uma maneira útil de localizar o arquivo a consultar quando você quiser ver o código que trata uma chamada de sistema em particular.

Por fim, *glo.h* (linha 19900) define as variáveis globais. Os buffers de mensagem para as mensagens que chegam e para as mensagens de resposta também estão aqui. O truque agora familiar com a macro *EXTERN* é utilizado, assim tais variáveis podem ser acessadas por todas as partes do sistema de arquivos. Como nas outras partes do MINIX o espaço de armazenamento será reservado quando *table.c* for compilado.

A parte do sistema de arquivos na tabela de processos está contida em *fproc.h* (linha 20000). A matriz *fproc* é declarada com a macro *EXTERN*. Ela armazena a máscara de modos, os ponteiros para os nós-i dos diretórios raiz e de trabalho atuais, a matriz de descritores de arquivo, *uid*, *gid* e número de terminal para cada processo. O *id* do processo e o *id* de grupo do processo também estão localizados aqui. Esses são duplicados em partes da tabela de processos localizada no *kernel* e no gerenciador de memória.

Vários campos são utilizados para armazenar os parâmetros das chamadas de sistema que podem ser suspensas no meio do caminho, como leituras de uma canalização vazia. Os campos *fp_suspended* e *fp_revived* requerem apenas bits simples, mas quase todos os compiladores geram código melhor para caracteres do que para campos de bit. Também há um campo para os bits *FD_CLOEXEC* indicados pelo padrão POSIX. Esses são utilizados para indicar que um arquivo deve ser fechado quando uma chamada EXEC é feita.

Agora chegamos aos arquivos que definem outras tabelas mantidas pelo sistema de arquivos. O primeiro, *buf.h* (linha 20100), define o *cache* de blocos. As estruturas aqui são todas declaradas com *EXTERN*. A matriz *buf* armazena todos os buffers, cada um dos quais contém uma parte de dados, *b*, e um cabeçalho cheio de ponteiros, de sinalizadores e de contadores. A parte de dados é declarada como uma união de cinco tipos (linha 20117) porque, às vezes, é conveniente referenciar o bloco como uma matriz de caracteres, outras vezes, como um diretório, etc.

A maneira adequada de referenciar a parte de dados do buffer 3 como uma matriz de caracteres é *buf*[3].*b.b__data* porque *buf[3].b* referencia a união como um todo, a partir da qual o campo *b__data* é selecionado. Embora essa sintaxe seja correta, ela é enfadonha; então, na linha 20142 definimos uma macro *b_ data* que nos permite escrever *buf*[3].*b_ data* no lugar. Note que *b__data* (o campo da união) contém dois sublinhados, ao passo que *b_ data* (a macro) contém somente um, para distingui-los. As macros para outras maneiras de acessar o bloco são definidas nas linhas 20143 até 20148.

A tabela de *hash* de buffers, *buf_hash*, é definida na linha 20150. Cada entrada aponta para uma lista de buffers. Originalmente todas as listas estão vazias. As macros no fim de *buf.h* (linhas 20160 a 20166) definem diferentes tipos de blocos. Quando um bloco é retornado para o *cache* de buffers depois de utilizado, um desses valores é fornecido para dizer ao gerenciador de *cache* se é para colocar o

bloco na frente ou no fundo da lista de LRU e se é para gravar em disco imediatamente ou não. O bit *WRITE_IMMED* sinaliza que um bloco deve ser regravado no disco imediatamente se for alterado. O superbloco é a única estrutura incondicionalmente marcada com isso. E quanto à outra estrutura marcada com *MAYBE_WRITE_IMMED*? Essa é definida em *include/minix/config.h* para ser igual a *WRITE_IMMED* se *ROBUST* for verdadeiro ou 0 caso contrário. Na configuração-padrão do MINIX, *ROBUST* é definido como 0 e esses blocos serão gravados quando blocos de dados forem gravados.

Por fim, na última linha (linha 20168) *HASH_MASK* é definida com base no valor de *NR_BUF_HASH* configurado em *include/minix/config.h*. *HASH_MASK* é "associada por meio da operação AND" com um número de bloco para determinar qual entrada em *buf_hash* utilizar como o ponto de partida para uma pesquisa por um buffer de bloco.

O próximo arquivo, *dev.h* (linha 20200) define a tabela *dmap*. A tabela em si é declarada em *table.c* com valores iniciais, assim tal versão não pode ser incluída em vários arquivos. Essa é a razão por que *dev.h* é necessário. *Dmap* é declarado aqui com *extern*, em vez de *EXTERN*. A tabela oferece o mapeamento entre o número de dispositivo principal e a tarefa correspondente.

File.h (linha 20300) contém a tabela intermediária *filp* (declarada como *EXTERN*), utilizada para armazenar a posição de arquivo atual e o ponteiro de nó-i (veja a Figura 5-33). Também diz se o arquivo foi aberto para leitura, para gravação ou para ambos e quantos descritores de arquivo atualmente estão apontando para a entrada.

A tabela de bloqueio de arquivo *file_lock* (declarada como *EXTERN*), está em *lock.h* (linha 20400). O tamanho da matriz é determinado por *NR_LOCKS*, que é definido como 8 em *const.h*. Esse número deve ser aumentado se for desejado implementar um banco de dados multiusuário em um sistema MINIX.

Em *inode.h* (linha 20500), a tabela de nós-i *inode* é declarada (utilizando *EXTERN*). Ela armazena os nós-i que estão atualmente em uso. Como dissemos anteriormente, quando um arquivo está aberto seu nó-i é carregado na memória e mantido aí até que o arquivo seja fechado. A definição de estrutura *inode* fornece as informações que são mantidas na memória, mas não gravadas no nó-i em disco. Note que há apenas uma versão e que nada é específico de versão aqui. Quando o nó-i é lido do disco, as diferenças entre os sistemas de arquivos V1 e V2 são tratadas. O restante do sistema de arquivos não necessita saber sobre o formato do sistema de arquivos no disco, pelo menos até chegar a hora de gravar de volta as informações modificadas.

A maioria dos campos deve ser auto-explicativa nesse ponto. Entretanto, *i_seek* merece algum comentário. Foi mencionado anteriormente que, como otimização, quando o sistema de arquivos nota que um arquivo está sendo lido seqüencialmente, ele tenta ler blocos no *cache* mesmo antes de eles serem pedidos. Para arquivos acessados aleatoriamente, não há nenhuma leitura antecipada. Quando uma chamada LSEEK é feita, o campo *i_seek* é ligado para inibir o buffer de leitura antecipada.

O arquivo *param.h* (linha 20600) é análogo ao arquivo de mesmo nome no gerenciador de memória. Ele define nomes para campos de mensagem contendo parâmetros, assim o código pode referenciar, por exemplo, buffer, em vez de *m.m1_p1*, o que seleciona um dos campos do buffer da mensagem *m*.

Em *super.h* (linha 20700) temos a declaração da tabela de superbloco. Quando o sistema é inicializado, o superbloco para o dispositivo-raiz é carregado aqui. Quando sistemas de arquivos são montados, seus superblocos entram aqui também. Como com outras tabelas, *super_block* é declarada como *EXTERN*.

Alocação de Armazenamento do Sistema de Arquivos

O último arquivo que discutiremos nesta seção não é um cabeçalho. Entretanto, como fizemos ao discutir o gerenciador de memória, parece apropriado discutir *table.c* imediatamente depois de revisar os arquivos de cabeçalho, uma vez que todos são incluídos quando *table.c* é compilado. Muitas das estruturas de dados que mencionamos — o *cache* de bloco, a tabela *filp* e assim por diante — são definidas com a macro *EXTERN*, assim como também são as variáveis globais do sistema de arquivos e a parte do sistema de arquivos da tabela de processos. Da mesma maneira como vimos em outras partes do sistema MINIX, o espaço de armazenamento é realmente reservado quando *table.c* é compilado. Esse arquivo também contém duas importantes matrizes inicializadas. *Call_vector* contém a matriz de ponteiros utilizada no laço principal para determinar qual procedimento trata qual número de chamada de sistema. Vimos uma tabela semelhante dentro do gerenciador de memória.

Uma coisa nova, entretanto, é a tabela *dmap* na linha 20914. Essa tabela tem uma linha para cada dispositivo principal, iniciando em zero. Quando um dispositivo é aberto, lido, fechado ou gravado, é essa tabela que oferece o nome do procedimento a chamar para tratar a operação. Todos esses procedimentos são localizados no espaço de endereço do sistema de arquivos. Muitos desses procedimentos não fazem nada, mas alguns chamam uma tarefa para de fato solicitar E/S. O número de tarefa correspondente a cada dispositivo principal também é oferecido pela tabela.

Sempre que um novo dispositivo principal é adicionado ao MINIX, uma linha deve ser adicionada a essa tabela informando que a ação, se alguma, deve ser tomada quando o dispositivo é aberto, fechado, lido ou gravado. Como um exemplo simples, se uma unidade de fita é adicionada ao MINIX, quando seu arquivo especial é aberto, o procedimento na tabela poderia verificar se a unidade de fita já está em utilização. Para poupar os usuários do trabalho de modificar essa tabela ao fazer a reconfiguração, uma ma-

cro, *DT*, é definida para automatizar o processo (linha 20900).

Há uma linha na tabela para cada possível dispositivo principal e cada linha é escrita com a macro. Dispositivos solicitados têm um 1 como o valor do argumento *enable* para a macro. Algumas entradas não são utilizadas, seja porque um *driver* planejado ainda não está pronto, seja porque um *driver* antigo foi removido. Essas entradas são definidas com um valor de 0 para *enable*. As entradas para dispositivos que podem ser configurados em *include/minix/config.h* utilizam a macro que ativa o dispositivo, por exemplo, *ENABLE_WINI* na linha 20920.

5.7.2 Gerenciamento de Tabelas

Associado a cada uma das tabelas principais — blocos, nós-i, superblocos, etc. — está um arquivo que contém procedimentos que gerenciam a tabela. Esses procedimentos são intensamente utilizados pelo restante do sistema de arquivos e formam a interface principal entre as tabelas e o sistema de arquivos. Por essa razão, é apropriado começar nosso estudo sobre o código do sistema de arquivos a partir desses procedimentos.

Gerenciamento de Blocos

O *cache* de blocos é gerenciado pelos procedimentos no arquivo *cache.c*. Esse arquivo contém os nove procedimentos listados na Figura 5-34. O primeiro, *get_block* (linha 21027), é a maneira-padrão como o sistema de arquivos obtém blocos de dados. Quando um procedimento do sistema de arquivos precisa ler um bloco de dados de usuário, um bloco de diretório, um superbloco ou qualquer outro tipo de bloco, ele chama *get_block*, especificando o dispositivo e o número de bloco.

Quando *get_block* é chamado, ele primeiro examina o *cache* de blocos para ver se o bloco solicitado está aí. Se estiver, retorna um ponteiro para ele. Caso contrário, precisa ler o bloco para a memória. Os blocos no *cache* estão encadeados juntos nas listas encadeadas *NR_BUF_HASH*. *NR_BUF_HASH* é um parâmetro ajustável, junto com *NR_BUFS*, o tamanho do *cache* de blocos. Esses dois são configurados em *include/minix/config.h*. No fim desta seção, diremos algumas palavras sobre otimizar o tamanho do *cache* de bloco e a tabela de *hash*. *HASH_MASK* é *NR_BUF_HASH* −1. Com 256 listas de *hash*, a máscara é 255, portanto, todos os blocos em cada lista têm números de bloco que acabam com a mesma cadeia de 8 bits, isto é, 00000000, 00000001, ..., ou 11111111.

Em geral, o primeiro passo é pesquisar um bloco na cadeia de *hash*, embora haja um caso especial, quando uma lacuna em um arquivo esparso está sendo lida, no qual essa pesquisa é pulada. Essa é a razão do teste na linha 21055. Caso contrário, as próximas duas linhas configuram *bp* para apontar para o início da lista em que o bloco solicitado estaria, se estivesse no *cache*, aplicando *HASH_MASK* ao número de bloco. O laço na próxima linha pesquisa essa lista para ver se o bloco pode ser localizado. Se for localizado e não estiver em utilização, o bloco é removido da lista de LRU. Se já estiver em utilização, ele não está na lista de LRU de qualquer maneira. O ponteiro para o bloco localizado é retornado para o chamador na linha 21063.

Se o bloco não está na lista de *hash*, ele não está no *cache*, então, o bloco menos recentemente utilizado da lista de LRU é tomado. O buffer escolhido é removido de sua cadeia de *hash*, uma vez que está prestes a receber um novo número de bloco e, portanto, pertence a uma cadeia de *hash* diferente. Se estiver sujo, ele é regravado em disco na linha 21095. Fazer isso com uma chamada a *flushall* regrava quaisquer outros blocos sujos do mesmo dispositivo. Os blocos que estão atualmente em utilização nunca são escolhidos para expulsão, uma vez que eles não estão na cadeia de LRU. Mas os blocos quase nunca serão encontrados como estando em utilização; normalmente um bloco é liberado por *put_block* imediatamente depois de ser utilizado.

Procedimento	Função
get_block	Buscar um bloco para ler ou gravar
put_block	Retornar um bloco previamente solicitado com get_block
alloc_zone	Alocar uma nova zona (para fazer um arquivo mais longo)
free_zone	Liberar uma zona (quando um arquivo é removido)
rw_block	Transferir um bloco entre disco e cache
invalidate	Purgar todo o *cache* de blocos para algum dispositivo
flushall	Gravar todos os blocos sujos de um dispositivo
rw_scattered	Ler ou gravar dados dispersos de ou para um dispositivo
rm_lru	Remover um bloco de sua cadeia de LRU

Figura 5-34 Os procedimentos utilizados para gerenciamento de blocos.

Logo que o buffer torna-se disponível, todos os campos, incluindo *b_dev*, são atualizados com os novos parâmetros (linhas 21099 a 21104) e o bloco pode ser lido para a memória a partir do disco. Entretanto, há duas ocasiões em que pode não ser necessário ler o bloco do disco. *Get_block* é chamado com um parâmetro *only_search*. Isso pode indicar que se trata de uma pré-busca. Durante uma pré-busca, um buffer disponível é localizado, gravando o conteúdo antigo em disco se necessário, e um novo número de bloco é atribuído ao buffer, mas o campo *b_dev* é configurado como *NO_DEV* para sinalizar que não há até agora quaisquer dados válidos nesse bloco. Veremos como isso é utilizado quando discutirmos a função *rw_scattered*. *Only_search* também pode ser utilizado para sinalizar que o sistema de arquivos necessita de um bloco simplesmente para regravá-lo inteiramente. Nesse caso é um desperdício ler primeiro a versão antiga. Em ambos os casos, os parâmetros são atualizados, mas a leitura real de disco é omitida (linhas 21107 até 21111). Quando o novo bloco for lido para a memória, *get_block* retorna para seu chamador com um ponteiro para ele.

Suponha que o sistema de arquivos precise de um bloco de diretório temporariamente para pesquisar um nome de arquivo. O sistema chama *get_block* para adquirir o bloco de diretório. Depois de pesquisar seu nome de arquivo, o sistema chama *put_block* (linha 21119) para retornar o bloco para o *cache*, tornando assim o buffer disponível no caso de esse ser necessário mais tarde para um bloco diferente.

Put_block cuida de colocar o bloco recentemente retornado na lista de LRU e, em alguns casos, de regravá-lo no disco. Na linha 21144, toma-se a decisão de colocá-lo na frente ou no fundo da lista de LRU, dependendo de *block_type*, um sinalizador fornecido pelo chamador e que informa o tipo do bloco. Os blocos que podem ser necessários novamente logo vão no final, assim eles permanecerão à mão por algum tempo. Os blocos que provavelmente não serão necessários novamente logo são colocados na frente, onde serão reutilizados de modo rápido. Atualmente, somente superblocos são assim tratados.

Depois que o bloco foi reposicionado na lista de LRU, outra verificação é feita (linhas 21172 e 21173) para ver se o bloco deve ser regravado em disco imediatamente. Na configuração-padrão, somente superblocos são marcados para gravação imediata, mas a única vez que um superbloco é modificado e precisa ser gravado é quando um disco de RAM é redimensionado na inicialização de sistema. Nesse caso, a gravação é para o disco de RAM e é improvável que o superbloco de um disco de RAM precise ser lido novamente. Assim, essa capacidade é dificilmente utilizada. Entretanto, a macro *ROBUST* em *include/minix/config.h* poder ser editada para indicar gravação imediata dos nós-i, dos blocos de diretório e de outros blocos que são essenciais ao funcionamento correto do próprio sistema de arquivos.

À medida que o arquivo cresce, de tempos em tempos uma nova zona deve ser alocada para armazenar os novos dados. O procedimento *alloc_zone* (linha 21180) cuida de alocar novas zonas. Ele faz isso localizando uma zona livre no mapa de bits de zonas. Não há nenhuma necessidade de pesquisar pelo mapa de bits se essa será a primeira zona em um arquivo; o campo *s_zsearch* no superbloco, que sempre aponta para a primeira zona disponível no dispositivo, é consultado. Caso contrário, é feita uma tentativa para localizar uma zona perto da última zona existente do arquivo atual para manter as zonas de um arquivo juntas. Isso é feito iniciando a pesquisa do mapa de bits a partir dessa última zona (linha 21203). O mapeamento entre o número do bit no mapa de bits e o número da zona é tratado na linha 21215, com o bit 1 correspondendo à primeira zona de dados.

Quando um arquivo é removido, suas zonas devem ser devolvidas ao mapa de bits. *Free_zone* (linha 21222) é responsável por devolver essas zonas. Tudo que faz é chamar *free_bit*, passando o mapa de zonas e o número do bit como parâmetros. *Free_bit* também é utilizado para retornar nós-i livres, mas, então, com o mapa de nós-i como o primeiro parâmetro, naturalmente.

O gerenciamento do *cache* requer leitura e gravação de blocos. Para proporcionar uma interface de disco simples, o procedimento *rw_block* (linha 21243) foi oferecido. Ele lê ou grava um bloco. Analogamente, *rw_inode* existe para ler e para gravar nós-i.

O próximo procedimento no arquivo é *invalidate* (linha 21280). Ele é chamado quando um disco é desmontado, por exemplo, para remover do *cache* todos os blocos pertencentes ao sistema de arquivos recentemente desmontado. Se isso não for feito, então, quando o dispositivo for reutilizado (com um disquete diferente), o sistema de arquivos talvez localize os blocos antigos em vez dos novos.

Flushall (linha 21295) é chamado pela chamada de sistema SYNC para descarregar para o disco todos os buffers sujos pertencentes a um dispositivo específico. Ele é chamado uma vez para cada dispositivo montado. Ele trata o *cache* de buffers como uma matriz linear, então, todos os buffers sujos são localizados, mesmo que estejam atualmente em utilização e não estejam na lista de LRU. Todos os buffers no *cache* são varridos e aqueles pertencentes ao dispositivo a ser descarregado e que precisam ser gravados são adicionados a uma matriz de ponteiros, *dirty*. Essa matriz é declarada como *static* para mantê-la fora da pilha. Então, ela é passada para *rw_scattered*.

Rw_scattered (linha 21313) recebe um identificador de dispositivo, um ponteiro para uma matriz de ponteiros para buffers, o tamanho da matriz e um sinalizador que indica se é para ler ou para gravar. A primeira coisa que ele faz é classificar a matriz que recebe pelos números de bloco, de modo que a operação de leitura ou de gravação seja executada em uma ordem eficiente. Ele é chamado com o sinalizador *WRITING* somente a partir da função *flushall* descrita anteriormente. Nesse caso, a origem desses números de bloco é fácil de entender. São buffers que contêm dados de blocos previamente lidos, mas agora modificados. A única chamada a *rw_scattered* para uma operação

de leitura é de *rahead* em *read.c*. Nesse ponto, apenas precisamos saber que antes de chamar *rw_scattered*, *get_block* foi chamado repetidamente em modo de pré-busca, reservando, assim, um grupo de buffers, os quais contêm números de bloco, mas nenhum parâmetro válido de dispositivo. Isso não é um problema, uma vez que *rw_scattered* é chamado com um parâmetro de dispositivo como um de seus argumentos.

Há uma diferença importante na maneira como o *driver* de dispositivo pode responder a uma solicitação de leitura (em oposição a uma gravação) de *rw_scattered*. Uma solicitação para gravar um número de blocos *deve* ser honrada completamente, mas uma solicitação para ler um número de blocos pode ser tratada diferentemente por *drivers* diferentes, dependendo do que é mais eficiente para o *driver* em particular. *Rahead* freqüentemente chama *rw_scattered* com uma solicitação para uma lista de blocos que realmente podem não ser necessários, assim, a melhor resposta é obter tantos blocos quanto puderem ser obtidos facilmente, mas não ir descontroladamente buscar tudo que há em um dispositivo que pode ter um tempo de busca substancial. Por exemplo, o *driver* de disquete pode parar em um limite de trilha e muitos outros *drivers* somente lerão blocos consecutivos. Quando a leitura está completa, *rw_scattered* marca a leitura de blocos, preenchendo o campo de número de dispositivo em seus buffers de bloco.

A última função na Figura 5-34 é *rm_lru* (linha 21387). Essa função é utilizada para remover um bloco da lista de LRU. Ela é utilizada somente por *get_block* nesse arquivo, então ela é declarada *PRIVATE* em vez de *PUBLIC* para ocultá-la de procedimentos fora do arquivo.

Antes de deixarmos o *cache* de blocos, precisamos dizer algumas palavras sobre o ajuste fino. *NR_BUF_HASH* dever ser uma potência de 2. Se for maior que *NR_BUFS*, o comprimento médio de uma cadeia de *hash* será menor que um. Se houver memória suficiente para um número grande de buffers, haverá espaço para um número grande de cadeias de *hash*, então, a escolha normal é tornar *NR_BUF_HASH* a próxima potência de 2 maior que *NR_BUFS*. A listagem no texto mostra configurações de 512 blocos e de 1024 listas de *hash*. O tamanho ótimo depende de como o sistema é utilizado, uma vez que isso determina quanto deve ser *bufferizado*. Empiricamente descobriu-se que aumentar o número de buffers além de 1024 não melhora o desempenho quando se recompila o sistema MINIX; portanto, aparentemente, isso é grande o suficiente para armazenar os binários para todas as passagens do compilador. Para algum outro tipo de trabalho um tamanho menor é adequado ou um tamanho maior talvez melhore o desempenho.

Os arquivos binários para o sistema MINIX no CD-ROM são compilados com um *cache* de blocos muito menor. A razão disso é que o binário de distribuição é destinado a executar em tantas máquinas quanto possível. Quis-se produzir uma versão de distribuição do MINIX que poderia ser instalada em um sistema, com somente 2MB de memória de RAM. Um sistema compilado com um *cache* de 1024 blocos requer mais de 2MB de RAM. O binário distribuído também inclui cada possível *driver* de disco rígido e outros *drivers* que podem não ser úteis em uma instalação particular. A maioria dos usuários vai querer editar *include/minix/config.h* e recompilar o sistema logo depois da instalação, omitindo *drivers* desnecessários e aumentando o *cache* de blocos, tanto quanto possível.

Enquanto no assunto do *cache* de blocos, indicaremos que o limite de 64KB no tamanho de segmento de memória em processadores Intel de 16 bits torna impossível um *cache* grande nessas máquinas. É possível configurar o sistema de arquivos para utilizar o disco de RAM como um *cache* secundário, armazenando blocos que são empurrados para fora do *cache* primário. Não discutimos isso aqui porque não é necessário em um sistema Intel de 32 bits; quando possível, um *cache* primário grande oferecerá melhor desempenho. Um *cache* secundário pode ser útil, entretanto, em uma máquina (como um 286) que não tem espaço para um *cache* primário grande dentro do espaço de endereço virtual do sistema de arquivos. Um *cache* secundário deve executar melhor que um disco convencional de RAM. Um *cache* armazena somente dados que são necessários pelo menos uma vez e, se suficientemente grande, pode fazer uma melhora grande no desempenho do sistema. O "suficientemente grande" não pode ser definido de antemão; somente pode ser medido vendo se aumentos em tamanho resultam em aumentos no desempenho. O comando *time*, que mede o tempo utilizado na execução de um programa, é uma ferramenta útil quando se tenta otimizar um sistema.

Gerenciamento de Nós-i

O *cache* de blocos não é a única tabela que precisa de procedimentos de suporte. A tabela de nós-i, também precisa. Muitos dos procedimentos são semelhantes em função aos procedimentos de gerenciamento de blocos. São listados na Figura 5-35.

O procedimento *get_inode* (linha 21534) é análogo a *get_block*. Quando qualquer parte do sistema de arquivos precisa de um nó-i, ele chama *get_inode* para obtê-lo. *Get_inode* primeiro pesquisa a tabela *inode* para ver se o nó-i já está presente. Se estiver, incrementa o contador de uso e retorna um ponteiro para ele. Essa pesquisa está contida nas linhas 21546 a 21556. Se o nó-i não estiver presente na memória, o nó-i é carregado, chamando *rw_inode*.

Quando o procedimento que precisava do nó-i termina, o nó-i é retornado, chamando o procedimento *put_inode* (linha 21578), que decrementa a contagem de utilização *i_count*. Se a contagem, então, é zero, o arquivo não está mais em utilização, e o nó-i pode ser removido da tabela. Se está sujo, é regravado para o disco.

Se o campo *i_link* é zero, nenhuma entrada de diretório está apontando para o arquivo, então, todas as suas zonas podem ser liberadas. Note que a contagem de utilização que está indo para zero e o número de vínculos indo

Procedimento	Função
get_inode	Buscar um nó-i na memória
put_inode	Retornar um nó-i que não é mais necessário
alloc_inode	Alocar um novo nó-i (para um novo arquivo)
wipe_inode	Limpar alguns campos em um nó-i
free_inode	Liberar um nó-i (quando um arquivo é removido)
update_times	Atualizar campos de tempo em um nó-i
rw_inode	Transferir um nó-i entre memória e disco
old_icopy	Converter o conteúdo de nó-i para gravar em nó-i de disco V1
new_icopy	Converter dados lidos do nó-i de disco de sistema de arquivos V1
dup_inode	Indicar que outra pessoa está utilizando um nó-i

Figura 5-35 Os procedimentos utilizados para gerenciamento de nós-i.

para zero são eventos diferentes, com causas diferentes e com conseqüências diferentes. Se o nó-i é para uma canalização, todas as zonas devem ser liberadas, mesmo que o número de vínculos possa não ser zero. Isso acontece quando um processo que lê de uma canalização libera a canalização.

Quando um novo arquivo é criado, um nó-i deve ser alocado por *alloc_inode* (linha 21605). O MINIX permite a montagem de dispositivos em modo apenas para leitura, então, o superbloco é verificado para assegurar que o dispositivo é gravável. Diferentemente das zonas, em que é feita uma tentativa de manter as zonas de um arquivo juntas, qualquer nó-i serve. Para poupar o tempo de pesquisar o mapa de bits de nós-i, tira-se proveito do campo no superbloco onde o primeiro nó-i não-utilizado foi registrado.

Depois que o nó-i foi adquirido, *get_inode* é chamado para buscar o nó-i na tabela que está na memória. Então, seus campos são inicializados, parcialmente *in-line* (linhas 21641 a 21648) e parcialmente utilizando o procedimento *wipe_inode* (linha 21664). Essa divisão particular de trabalho foi escolhida porque *wipe_inode* também é necessário em outras partes no sistema de arquivos para limpar certos campos de nó-i (mas não todos eles).

Quando um arquivo é removido, seu nó-i é liberado chamando *free_inode* (linha 21684). Tudo o que acontece aqui é que o bit correspondente no mapa de bits de nós-i é configurado como 0, e o registro do superbloco do primeiro nó-i não-utilizado é atualizado.

A próxima função, *update_times* (linha 21704), é chamada para obter o tempo do relógio de sistema e para alterar os campos de tempo que precisarem ser atualizados. *Update_times* também é chamada pelas chamadas de sistema STAT e FSTAT, então, ela é declarada *PUBLIC*.

O procedimento *rw_inode* (linha 21731) é análogo a *rw_block*. Seu trabalho é buscar um nó-i do disco. Faz seu trabalho executando os seguintes passos:

1. Calcular qual bloco contém o nó-i solicitado.
2. Ler o bloco chamando *get_block*.
3. Extrair o nó-i e copiá-lo para a tabela *inode*.
4. Retornar o bloco, chamando *put_block*.

Rw_inode é um pouco mais complexo que o esboço básico dado acima, então, algumas funções adicionais são necessárias. Primeiro, como obter o tempo atual é caro, qualquer necessidade de alterar os campos de tempo no nó-i é apenas marcada, ligando-se bits no campo *i_update* do nó-i enquanto o nó-i está na memória. Se esse campo for diferente de zero quando um nó-i precisar ser gravado, *update_times* será chamado.

Em segundo lugar, a história do MINIX adiciona uma complicação: na antiga versão, *V1*, do sistema de arquivos os nós-i no disco tinham uma estrutura diferente da *V2*. Duas funções, *old_icopy* (linha 21774) e *new_icopy* (linha 21821) cuidam das conversões. A primeira faz a conversão entre as informações de nó-i na memória e o formato utilizado pelo sistema de arquivos *V1*. A segunda faz a mesma conversão para discos com sistema de arquivos *V2*. Ambas as funções são chamadas somente a partir desse arquivo, então, são declaradas *PRIVATE*. Cada função trata as conversões em ambas as direções (do disco para a memória ou da memória para o disco). O MINIX foi implementado em sistemas, utilizando uma ordem de byte diferente dos processadores Intel. Cada implementação utiliza a ordem de byte nativa em seu disco; o campo *sp->native* no superbloco identifica que ordem é utilizada. Ambas, *old_icopy* e *new_icopy*, chamam as funções *conv2* e *conv4* para trocar as ordens de byte, se necessário.

O procedimento *dup_inode* (linha 21865) simplesmente incrementa a contagem de utilização do nó-i. Ele é chamado quando um arquivo aberto é aberto novamente. Na segunda abertura, o nó-i não precisa ser buscado no disco novamente.

Gerenciamento de Superbloco

O arquivo *super.c* contém procedimentos que gerenciam o superbloco e os mapas de bits. Há cinco procedimentos nesse arquivo, listados na Figura 5-36.

Quando um nó-i ou uma zona é necessário, *alloc_inode* ou *alloc_zone* é chamada, como vimos acima. Ambas chamam *alloc_bit* (linha 21926) para realmente pesquisar o mapa de bits relevante. A pesquisa envolve três laços aninhados como segue:

1. O externo faz um laço em todos os blocos de um mapa de bits.
2. O do meio faz um laço em todas as palavras de um bloco.
3. O interno faz um laço em todos os bits de uma palavra.

O laço do meio funciona vendo se a palavra atual é igual ao complemento de um de zero, isto é, uma palavra completa cheia de 1s. Se for, ela não tem nós-i ou zonas livres, então, a próxima palavra é tentada. Quando uma palavra com um valor diferente é localizada, ela deve ter pelo menos um bit 0 nela; então, o laço interno é iniciado para localizar o bit livre (i. e., 0). Se todos os blocos forem tentados sem êxito, não há nós-i ou zonas livres, então, o código *NO_BIT* (0) é retornado. Pesquisas como essa podem consumir muito tempo do processador, mas o uso dos campos no superbloco que apontam para o primeiro nó-i e zona não-utilizados, passado para *alloc_bit* em *origin*, ajuda a manter essas pesquisas curtas.

Liberar um bit é mais simples que alocar, porque nenhuma pesquisa é necessária. *Free_bit* (linha 22003) calcula qual bloco do mapa de bits contém o bit a liberar e configura o bit adequado como 0 chamando *get_block*, para zerar o bit na memória e, então, chamar *put_block*.

O próximo procedimento, *get_super* (linha 22047), é utilizado para pesquisar a tabela de superbloco para um dispositivo específico. Por exemplo, quando um sistema de arquivos estiver para ser montado, é necessário verificar se ele já não está montado. Essa verificação pode ser executada solicitando a *get_super* para localizar o dispositivo do sistema de arquivos. Se o dispositivo não for localizado, então, o sistema de arquivos não está montado.

A próxima função, *mounted* (linha 22067), é chamada somente quando um dispositivo de bloco é fechado. Normalmente, todos os dados em *cache* para um dispositivo são descartados quando ele é fechado. Mas, se acontecer de o dispositivo estar montado, isso não é desejável. *Mounted* é chamada com um ponteiro para o nó-i de um dispositivo. Ela simplesmente retorna *TRUE* se o dispositivo for o dispositivo-raiz ou se for um dispositivo montado.

Por fim, temos *read_super* (linha 22088). Essa é parcialmente análoga a *rw_block* e *rw_inode*, mas é chamada somente para ler. A gravação de um superbloco não é necessária na operação normal do sistema. *Read_super* verifica a versão do sistema de arquivos do qual acaba de ler e executa conversões, se necessário; então, a cópia do superbloco na memória terá a estrutura-padrão mesmo quando lida de um disco com uma diferente estrutura de superbloco.

Gerenciamento de Descritores de Arquivos

O MINIX contém procedimentos especiais para gerenciar descritores de arquivos e a tabela *filp* (veja a Figura 5-33). Eles estão contidos no arquivo *filedes.c*. Quando um arquivo é criado ou aberto, um descritor de arquivo e uma entrada *filp* livres são necessários. O procedimento *get_fd* (linha 22216) é utilizado para localizá-los. Entretanto, eles não são marcados como em utilização, porque muitas verificações devem ser feitas antes de saber-se com certeza que CREAT ou OPEN tiveram sucesso.

Get_filp (linha 22263) é utilizada para ver se um descritor de arquivos está ao alcance e, se estiver, retorna seu ponteiro *filp*.

O último procedimento nesse arquivo é *find_filp* (linha 22277). Ele é necessário para saber quando um processo está gravando em uma canalização quebrada (i. e., em uma canalização não-aberta para leitura por qualquer outro processo). Ele localiza leitores potenciais por uma pesquisa de força bruta da tabela *filp*. Se não puder localizar um, a canalização é quebrada, e a gravação falha.

Bloqueio de Arquivos

As funções de bloqueio de registro do POSIX são mostradas na Figura 5-37. Uma parte de um arquivo pode ser bloqueada para leitura e gravação ou para gravação somente, por uma chamada FCNTL, especificando uma solicitação *F_SETLK* ou *F_SETLKW*. Se existe ou não um bloqueio so-

Procedimento	Função
alloc_bit	Alocar um bit do mapa de zonas ou de nós-i
free_bit	Liberar um bit do mapa de zonas ou de nós-i
get_super	Procurar a tabela de superbloco para um dispositivo
mounted	Informar se um nó-i dado está em um sistema de arquivos montado (ou raiz)
read_super	Ler um superbloco

Figura 5-36 Os procedimentos utilizados para gerenciar o superbloco e os mapas de bits.

Operação	Significado
F_SETLK	Bloqueia região tanto para leitura quanto para gravação
F_SETLKW	Bloqueia região para gravação
F_GETLK	Informa se a região está bloqueada

Figura 5-37 Operações de bloqueio de registro consultivas do POSIX. Essas operações são solicitadas utilizando a chamada de sistema FCNTL.

bre uma parte de um arquivo, isso pode ser determinado, utilizando a solicitação *F_GETLK*.

Há somente duas funções no arquivo *lock.c*. *Lock_op* (linha 22319) é chamada pela chamada de sistema FCNTL com um código para uma das operações mostradas na Figura 5-37. Ela faz alguma verificação de erro para certificar-se de que a região especificada é válida. Quando um bloqueio está sendo configurado, ele não deve conflitar um bloqueio existente; e quando um bloqueio está sendo limpo, um bloqueio existente não deve ser dividido em dois. Quando qualquer bloqueio é limpo, a outra função nesse arquivo, *lock_revive* (linha 22463), é chamada. Ela desperta todos os processos que estão bloqueados, esperando bloqueios. Essa estratégia é um acordo; tomaria código extra para descobrir exatamente quais processos estavam esperando um bloqueio particular ser liberado. Esses processos que ainda estão esperando um arquivo bloqueado bloquearão novamente quando iniciarem. Essa estratégia é baseada em uma suposição de que o bloqueio será utilizado raramente. Se um banco de dados multiusuário importante tivesse de ser construído sobre um sistema MINIX, talvez fosse desejável reimplementar isso.

Lock_revive também é chamada quando um arquivo bloqueado é fechado, como talvez aconteça, por exemplo, se um processo é eliminado antes de terminar de utilizar um arquivo bloqueado.

5.7.3 Programa Principal

O laço principal do sistema de arquivos está contido no arquivo *main.c*, iniciando na linha 22537. Estruturalmente, ele é muito semelhante ao laço principal do gerenciador de memória e das tarefas de E/S. A chamada a *get_work* espera a próxima mensagem de solicitação chegar (a menos que um processo previamente suspenso em uma canalização ou terminal agora possa ser tratado). Ele também configura uma variável global, *who*, para o número de entrada da tabela de processos do chamador e outra variável global, *fs_call*, para o número da chamada de sistema a ser executada.

Uma vez de volta ao laço principal, três sinalizadores são configurados: *fp* aponta para a entrada da tabela de processos do chamador, *super_user* diz se o chamador é o superusuário ou não e *dont_reply* é inicializado como *FALSE*. Então, vem a atração principal — a chamada ao procedimento que executa a chamada de sistema. O procedimento a chamar é selecionado, utilizando *fs_call* como um índice na matriz de ponteiros de procedimento, *call_vector*.

Quando o controle volta para o laço principal, se *dont_reply* tiver sido configurado, a resposta é inibida (p. ex., um processo bloqueou tentando ler de uma canalização vazia). Caso contrário, uma resposta é enviada. A declaração final no laço principal foi projetada para detectar que um arquivo está sendo lido seqüencialmente e carregar o próximo bloco no *cache* antes de ele ser realmente solicitado para melhorar o desempenho.

O procedimento *get_work* (linha 22572) verifica se qualquer procedimento previamente bloqueado agora foi reanimado. Se tiver sido, esse tem prioridade sobre novas mensagens. Somente se não houver trabalho interno a fazer é que o sistema de arquivos chama o *kernel* para obter uma mensagem na linha 22598.

Depois que uma chamada de sistema foi completada, com êxito ou não, uma resposta é enviada de volta para o chamador por *reply* (linha 22608). O processo pode ter sido eliminado por um sinal, então, o código de status retornado pelo *kernel* é ignorado. Nesse caso, não há nada a ser feito de qualquer jeito.

Funções de Inicialização

O restante de *main.c* consiste em funções que são utilizadas somente na inicialização do sistema. Antes de o sistema de arquivos entrar em seu laço principal, ele se inicializa chamando *fs_init* (linha 22625) que, por sua vez, chama várias outras funções para inicializar o *cache* de blocos, para obter os parâmetros de inicialização, para carregar o disco de RAM se necessário e para carregar o superbloco do dispositivo-raiz. O próximo passo é inicializar a parte do sistema de arquivos na tabela de processos para todas as tarefas e servidores até *init* (linhas 22643 a 22654). Por fim, testes são feitos em algumas constantes importantes, para ver se fazem sentido, e uma mensagem é enviada para a tarefa de memória com o endereço da parte do sistema de arquivos na tabela de processos para utilização pelo programa *ps*.

A primeira função chamada por *fs_init* é *buf_pool* (linha 22679), que constrói as listas encadeadas utilizadas pelo *cache* de blocos. A Figura 5-31 mostra o estado normal do *cache* de blocos, em que todos os blocos estão encadeados, tanto na cadeia de LRU como na cadeia de *hash*.

Ele pode ser útil para ver como a situação da Figura 5-31 realiza-se. Imediatamente depois de o *cache* ser inicializado por *buf_pool*, todos os buffers estarão na cadeia de LRU e todos serão encadeados na cadeia 0 de *hash*, como na Figura 5-38(a). Quando um buffer é solicitado e enquanto estiver em utilização, temos a situação da Figura 5-38(b), em que vemos que um bloco foi removido da cadeia de LRU e está agora em uma cadeia de *hash* diferente. Normalmente, os blocos são liberados e retornados à cadeia de LRU imediatamente. A Figura 5-38(c) mostra a situação depois que o bloco foi retornado à cadeia de LRU. Embora não esteja mais em utilização, ele pode ser acessado novamente para oferecer os mesmos dados, se for necessário, e, então, é retido na cadeia de *hash*. Depois que o sistema esteve em operação por algum tempo, provavelmente quase todos os blocos terão sido utilizados e distribuídos aleatoriamente entre as diferentes cadeias de *hash*. Então, a cadeia de LRU ficará parecida com a Figura 5-31.

A próxima função é *get_boot_parameters* (linha 22706). Ela envia uma mensagem à tarefa de sistema, pedindo-lhe uma cópia dos parâmetros de inicialização. Esses são necessários para a função seguinte, *load_ram* (linha 22722), que aloca espaço para um disco de RAM. Se os parâmetros de inicialização especificarem

```
rootdev = ram
```

o sistema de arquivos do dispositivo-raiz será copiado do dispositivo nomeado por *ramimagedev* para o disco de RAM, bloco por bloco, iniciando com o bloco de inicialização, sem interpretação das várias estruturas de dados do sistema de arquivos. Se o parâmetro de inicialização *ramsize* for menor que o tamanho do sistema de arquivos do

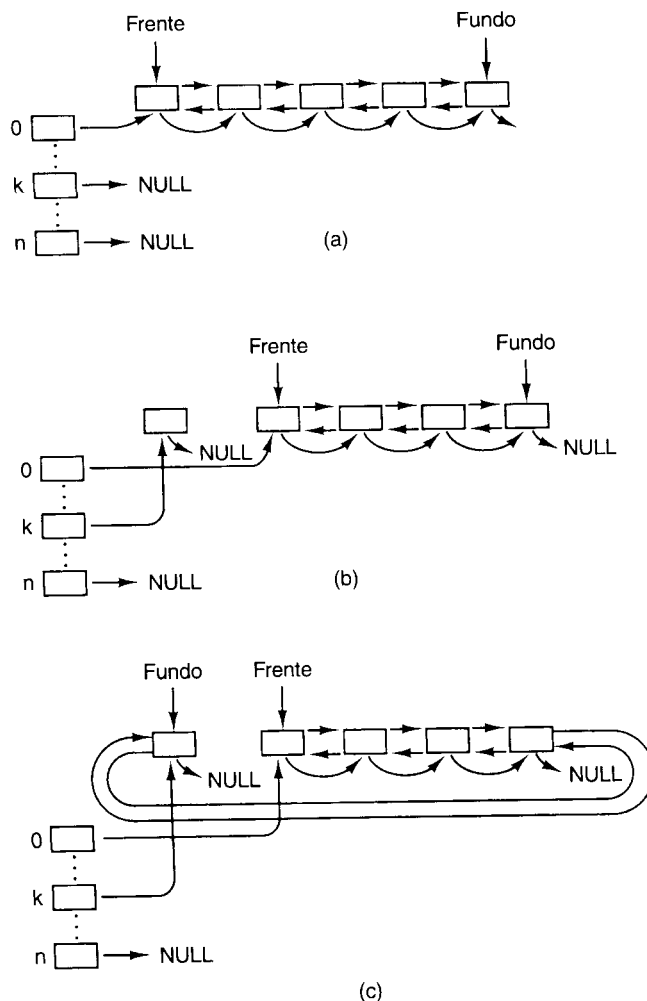



Figura 5-38 Inicialização do *cache* de blocos. (a) Antes de qualquer buffer ser utilizado. (b) Depois que um bloco foi solicitado. (c) Depois que o bloco foi liberado.

dispositivo-raiz, o disco de RAM será feito grande o suficiente para armazená-lo. Se *ramsize* especificar um tamanho maior que o sistema de arquivos do dispositivo de inicialização, o tamanho especificado será alocado, e o sistema de arquivos do disco de RAM será ajustado para utilizar inteiramente o tamanho especificado (linhas 22819 a 22825) A chamada a *put_block* na linha 22825 é o único momento em que o sistema de arquivos grava um superbloco.

Load_ram aloca espaço para um disco de RAM vazio se um *ramsize* não-zero for especificado. Nesse caso, uma vez que nenhuma estrutura do sistema de arquivos foi copiada, o dispositivo de RAM não pode ser utilizado como um sistema de arquivos até que seja inicializado pelo comando *mkfs*. Alternativamente, esse disco de RAM poderá ser utilizado para um *cache* secundário, se o suporte para isso for compilado no sistema de arquivos.

A última função em *main.c* é *load_super* (linha 22832). Ela inicializa a tabela de superbloco, e carrega o superbloco do dispositivo-raiz.

5.7.4 Operações em Arquivos Individuais

Nesta seção, veremos as chamadas de sistema que operam em arquivos individuais um por vez (em oposição a, digamos, operações em diretórios). Iniciaremos com a maneira como os arquivos são criados, abertos e fechados. Depois, examinaremos em algum detalhe o mecanismo pelo qual os arquivos são lidos e gravados. Em seguida, veremos como as canalizações e as operações nelas diferem daquelas em arquivos.

Criando, Abrindo e Fechando Arquivos

O arquivo *open.c* contém o código para seis chamadas de sistema: CREAT, OPEN, MKNOD, MKDIR, CLOSE e LSEEK. Examinaremos CREAT e OPEN juntas e, então, veremos cada uma da outras.

Em versões mais antigas do UNIX, as chamadas CREAT e OPEN tinham propósitos distintos. Tentar abrir um arquivo que não existia era um erro e um novo arquivo tinha de ser criado com CREAT, que também poderia ser utilizada para truncar um arquivo existente para comprimento zero. A necessidade de duas chamadas distintas não está mais presente em sistemas POSIX, entretanto sob o POSIX, a chamada OPEN agora permite criar um novo arquivo ou truncar um arquivo antigo, então, a chamada CREAT agora representa um subconjunto das possíveis utilizações da chamada OPEN e é realmente necessária apenas para compatibilidade com programas mais antigos. Os procedimentos que tratam CREAT e OPEN são *do_creat* (linha 22937) e *do_open* (linha 22951). (Como no gerenciador de memória, no sistema de arquivos é utilizada a convenção de que a chamada de sistema XXX é executada pelo procedimento *do_xxx*.) A abertura ou a criação de um arquivo envolve três passos:

1. Localizar o nó-i (alocar e inicializar se o arquivo for novo).
2. Localizar ou criar a entrada de diretório.
3. Configurar e retornar um descritor de arquivo para o arquivo.

Ambas as chamadas CREAT e OPEN fazem duas coisas: buscam o nome de um arquivo e, então, chamam *common_open* para cuidar das tarefas comuns a ambas as chamadas.

Common_open (linha 22975) inicia certificando-se de que entradas livres da tabela de descritores de arquivos e da tabela *filp* estão disponíveis. Se a função chamada especificou a criação de um novo arquivo (chamando com o bit *O_CREAT* ligado), *new_node* é chamada na linha 22998. *New_node* retornará um ponteiro para um nó-i existente se a entrada de diretório já existir; caso contrário, criará tanto uma nova entrada de diretório como o nó-i. Se o nó-i não puder ser criado, *new_node* configura a variável global *err_code*. Um código de erro nem sempre significa um erro. Se *new_node* localizar um arquivo existente, o código de erro retornado indicará que o arquivo existe, mas nesse caso esse erro é aceitável (linha 23001). Se o bit *O_CREAT* não estiver ligado, é feita uma pesquisa do nó-i, utilizando um método alternativo, a função *eat_path* em *path.c*, que posteriormente discutiremos. Nesse ponto, o importante é entender que se um nó-i não for localizado nem puder ser criado com êxito, *common_open* terminará com um erro antes de ter alcançado a linha 23010. Caso contrário, a execução continua aqui com a atribuição de um descritor de arquivo e reivindicando uma entrada na tabela *filp*. Seguindo-se a isso, se um novo arquivo acabou de ser criado, as linhas 23017 a 23094 são puladas.

Se o arquivo não for novo, então, o sistema de arquivos deve testar para ver de que tipo o arquivo é, seu modo e assim por diante, a fim de determinar se ele pode ser aberto. A chamada a *forbidden* na linha 23018 primeiro faz uma verificação geral dos bits *rwx*. Se o arquivo for comum e *common_open* for chamado com o bit *O_TRUNC* ligado, ele é truncado para comprimento zero e *forbidden* é chamada novamente (linha 23024), dessa vez para assegurar que o arquivo pode ser gravado. Se os direitos permitirem, *wipe_inode* e *rw_inode* são chamados para reinicializar o nó-i e gravar no disco. Outros tipos de arquivo (diretórios, arquivos especiais e canalizações nomeadas) são submetidos aos testes apropriados. No caso de um dispositivo, é feita uma chamada na linha 23053 (utilizando a estrutura *dmap*) à rotina apropriada para abrir o dispositivo. No caso de uma canalização nomeada, é feita uma chamada a *pipe_open* (linha 23060) e são feitos vários testes pertinentes.

O código de *common_open*, assim como muitos outros procedimentos do sistema de arquivos, contém uma quantidade grande de código que verifica vários erros e combinações ilegais. Embora não-fascinante, esse código é essencial para ter um sistema de arquivos livre de erros e

robusto. Se algo estiver errado, o descritor de arquivo e o slot *filp* previamente alocado são desalocados, e o nó-i é liberado (linhas 23098 a 23101). Nesse caso, o valor retornado por *common_open* será um número negativo, indicando um erro. Se não houver nenhum problema, o descritor de arquivo, que é um valor positivo, será retornado.

Esse é um bom lugar para discutir em mais detalhes a operação de *new_node* (linha 23111), que faz a alocação do nó-i e a entrada do nome de caminho no sistema de arquivos para as chamadas CREAT e OPEN. Ela também é utilizada para as chamadas MKNOD e MKDIR, ainda a serem discutidas. A declaração na linha 23128 analisa sintaticamente o nome de caminho (i. e., pesquisa-o componente por componente) até o diretório final; a chamada a *advance,* três linhas mais adiante, tenta ver se o componente final pode ser aberto.

Por exemplo, na chamada

```
fd = creat("/usr/ast/foobar", 0755);
```

last_dir tenta carregar o nó-i para */usr/ast* nas tabelas e retorna um ponteiro para ele. Se o arquivo não existir, necessitaremos desse nó-i brevemente para adicionar *foobar* ao diretório. Todas as outras chamadas de sistema que adicionam ou excluem arquivos também utilizam *last_dir* para primeiro abrir o diretório final no caminho.

Se *new_node* descobre que o arquivo não existe, ela chama *alloc_inode* na linha 23134 para alocar e carregar um novo nó-i, retornando um ponteiro para ele. Se não restar nenhum nó-i livre, *new_node* falha e retorna *NIL_INODE.*

Se um nó-i puder ser alocado, a operação continuará na linha 23144, preenchendo alguns campos, gravando-os de volta em disco e inserindo o nome de arquivo no diretório final (linha 23149). Novamente vemos que o sistema de arquivos deve constantemente verificar novos erros e, ao encontrar um, cuidadosamente liberar todos os recursos, como nós-i e blocos que ele está segurando. Se precisássemos simplesmente preparar o MINIX para gerar uma pane quando esgotássemos, digamos, os nós-i, em vez de desfazer todos os efeitos da chamada atual e retornar um código de erro para o chamador, o sistema de arquivos seria consideravelmente mais simples.

Como mencionado acima, as canalizações exigem tratamento especial. Se não houver pelo menos um par leitor/gravador para uma canalização, *pipe_open* (linha 23176) suspende o chamador. Caso contrário, chama *release,* que procura na tabela de processos por processos que estão bloqueados na canalização. Se for bem-sucedida, os processos são reanimados.

A chamada MKNOD é tratada por *do_mknod* (linha 23205). Esse procedimento é semelhante a *do_creat,* exceto que termina de criar o nó-i e faz uma entrada de diretório para ele. De fato, a maior parte do trabalho é feita pela chamada a *new_node* na linha 23217. Se o nó-i já existir, um código de erro será retornado. Esse é o mesmo código de erro que era um resultado aceitável de *new_node* quando foi chamado por *common_open*; nesse caso, entretanto, o código de erro é passado de volta para o chamador, que presumivelmente agirá assim. A análise caso a caso que vimos em *common_open* não é necessária aqui.

A chamada MKDIR é tratada pela função *do_mkdir* (linha 23226). Como com as outras chamadas de sistema que discutimos aqui, *new_node* desempenha um papel importante. Os diretórios, diferentemente dos arquivos, sempre têm vínculos e nunca estão completamente vazios porque cada diretório deve conter duas entradas criadas no momento da sua criação: as entradas "." e "..", que se referem ao próprio diretório e ao seu diretório-pai. Há um limite para o número de vínculos que um arquivo pode ter, *LINK_MAX* (definida em *include/limits.h* como 127 para o sistema MINIX padrão). Uma vez que a referência a um diretório-pai em um filho é um vínculo para o pai, a primeira coisa que *do_mkdir* faz é ver se é possível fazer outro vínculo no diretório-pai (linha 23240). Uma vez que esse teste foi passado, *new_node* é chamado. Se *new_node* for bem-sucedido, então, as entradas de diretório para "." e ".." são feitas (linhas 23261 e 23262). Tudo isso é simples e direto, mas poderia haver falhas (p. ex., se o disco estiver cheio) e para evitar confusão providenciou-se a possibilidade de desfazer as etapas iniciais do processo, caso elas não puderem ser completadas.

Fechar um arquivo é mais fácil que abrir. O trabalho é feito por *do_close* (linha 23286). As canalizações e os arquivos especiais necessitam de alguma atenção, mas para arquivos comuns, quase tudo que precisa ser feito é decrementar o contador *filp* e verificar se ele é zero, caso em que o nó-i é retornado com *put_inode.* O passo final é remover quaisquer bloqueios e resumir qualquer processo que pode ter sido suspenso, esperando um bloqueio no arquivo ser liberado.

Note que retornar um nó-i significa que seu contador na tabela *inode* é decrementado, de modo que ele possa eventualmente ser removido da tabela. Essa operação não tem nada a ver com liberar o nó-i (i. e., ligar um bit no mapa de bits que diz que ele está disponível). O nó-i somente é liberado quando o arquivo é removido de todos os diretórios.

O procedimento final em *open.c* é *do_lseek* (linha 23367). Quando uma busca é feita, esse procedimento é chamado para configurar a posição de arquivo para um novo valor. Na linha 23394, a leitura antecipada é inibida; uma tentativa explícita de buscar uma posição em um arquivo é incompatível com o acesso seqüencial.

Lendo um Arquivo

Uma vez que um arquivo foi aberto, ele pode ser lido ou gravado. Muitas funções são utilizadas tanto durante a leitura como durante a gravação. Essas funções estão localizadas no arquivo *read.c.* Discutiremos essas primeiro e, então, prosseguiremos para o arquivo seguinte, *write.c,* para ver o código especificamente utilizado para gravação. Leitura e gravação diferem sob vários aspectos, mas têm tantas semelhanças que tudo que é solicitado de *do_read* (li-

nha 23434) é chamar o procedimento comum *read_write* com um sinalizador configurado como *READING*. Veremos na próxima seção que *do_write* é igualmente simples.

Read_write começa na linha 23443. Há algum código especial nas linhas 23459 a 23462 que é utilizado pelo gerenciador de memória para fazer o sistema de arquivos carregar segmentos inteiros no espaço de usuário para ele. Chamadas normais são processadas, iniciando na linha 23464. Alguma verificação de validade segue-se (p. ex., leitura de um arquivo aberto somente para escrita) e algumas variáveis são inicializadas. As leituras de arquivos especiais de caractere não seguem pelo *cache* de blocos, assim, são filtradas na linha 23498.

Os testes nas linhas 23507 a 23518 aplicam-se somente a gravações e têm a ver com arquivos que podem ficar maiores do que o dispositivo pode armazenar, ou gravações que criarão uma lacuna no arquivo, gravando para além do fim do arquivo. Como discutimos na visão geral do MINIX, a presença de múltiplos blocos por zona causa problemas que devem ser tratados explicitamente. As canalizações também são especiais e são verificadas.

O núcleo do mecanismo de leitura, pelo menos para arquivos comuns, é o laço que inicia na linha 23530. Esse laço quebra a solicitação em pedaços, cada um dos quais se ajusta em um único bloco de disco. Um pedaço começa na posição atual e estende-se até que uma das seguintes condições seja satisfeita:

1. Todos os bytes foram lidos.
2. Um limite de bloco foi encontrado.
3. O fim do arquivo é atingido.

Essas regras significam que um pedaço nunca requer dois blocos de disco para satisfazê-lo. A Figura 5-39 mostra três exemplos de como o tamanho do pedaço é determinado, para tamanhos de 6, de 2 e de 1 bytes, respectivamente. O cálculo real é feito nas linhas 23632 a 23641.

A leitura real do pedaço é feita por *rw_chunk*. Quando o controle retorna, vários contadores e ponteiros são incrementados e, a próxima iteração começa. Quando o laço termina, a posição de arquivo e outras variáveis podem ser atualizadas (p. ex., ponteiros de canalizações).

Por fim, se a leitura antecipada for chamada, o nó-i e a posição a serem lidos são armazenados em variáveis globais, de modo que depois que a mensagem de resposta é enviada para o usuário, o sistema de arquivos pode começar a trabalhar para obter o próximo bloco. Em muitos casos, o sistema de arquivos bloqueará, esperando o próximo bloco de disco, tempo durante o qual o processo de usuário será capaz de trabalhar nos dados que acabou de receber. Esse arranjo sobrepõe-se ao processamento e à E/S e pode melhorar o desempenho substancialmente.

O procedimento *rw_chunk* (linha 23613) está preocupado em tomar um nó-i e uma posição de arquivo, convertê-los em um número físico de bloco de disco e solicitar a transferência desse bloco (ou uma parte dele) para o espaço do usuário. O mapeamento da posição de arquivo relativa ao endereço físico de disco é feito por *read_map*, que entende de nós-i e de blocos indiretos. Para um arquivo comum, as variáveis *b* e *dev* nas linhas 23637 e 23638 contêm o número físico do bloco e o número de dispositivo, respectivamente. A chamada a *get_block* na linha 23660 é onde o manipulador de *cache* é solicitado a localizar o bloco, lendo-o se necessário.

Uma vez que temos um ponteiro para o bloco, a chamada a *sys_copy* na linha 23670 cuida de transferir a parte solicitada dele para o espaço do usuário. O bloco, então, é liberado por *put_block*, de modo que pode ser expulso do *cache* mais tarde, quando chegar a hora. (Depois de ser adquirido por *get_block*, ele não estará na fila de LRU e não será retornado aí enquanto o contador no cabeçalho do bloco mostrar que ele está em utilização, portanto, ele ficará isento de expulsão; *put_block* decrementa o conta-

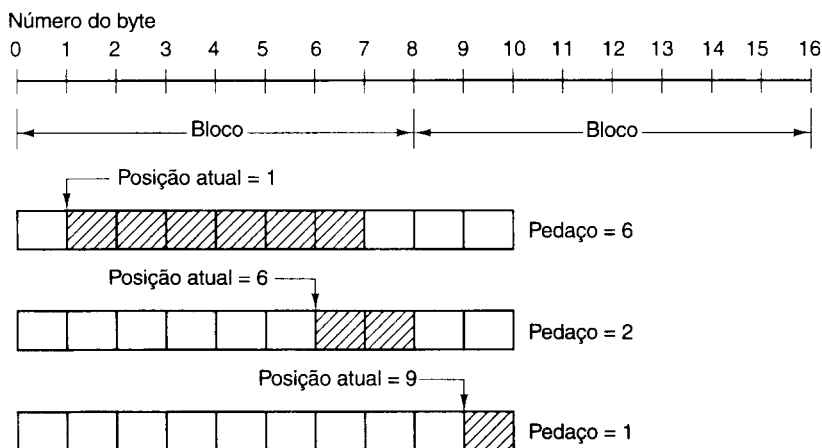



Figura 5-39 Três exemplos de como o primeiro tamanho de pedaço é determinado para um arquivo de 10 bytes. O tamanho de bloco é 8 bytes e o número de bytes solicitado é 6. O pedaço é mostrado sombreado.

dor e retorna o bloco à fila de LRU quando o contador atingir zero.) O código na linha 23680 indica se uma operação de gravação preencheu o bloco. Entretanto, o valor passado para *put_block* em *n* não afeta como o bloco é colocado na fila; todos os blocos agora estão colocados no fim da cadeia de LRU.

Read_map (linha 23689) converte uma posição lógica de arquivo para o número físico de bloco inspecionando o nó-i. Para blocos próximos o suficiente do começo do arquivo, eles caem dentro de uma das primeiras sete zonas (aquele exatamente no nó-i), um cálculo simples é suficiente para determinar qual zona é necessária e então qual bloco. Para blocos adiante no arquivo, um ou mais blocos indiretos podem precisar ser lidos.

Rd_indir (linha 23753) é chamado para ler um bloco indireto. Ele foi feito como um procedimento separado porque há diferentes formatos que os dados podem assumir no disco, dependendo da versão do sistema de arquivos e do hardware em que o sistema de arquivos foi escrito. As confusas conversões são feitas aqui, se necessário; então, o restante do sistema de arquivos vê os dados em apenas um formato.

Read_ahead (linha 23786) converte a posição lógica para um número físico de bloco, chama *get_block* para certificar-se de que o bloco está no *cache* (ou o traz) e, então, retorna o bloco imediatamente. Esse procedimento não pode fazer nada com o bloco, afinal de contas. Apenas quer melhorar a chance de que o bloco esteja à mão se este último precisar ser utilizado em seguida.

Note que *read_ahead* é chamado somente a partir do laço principal em *main*. Ele não é chamado como parte do processamento da chamada de sistema READ. É importante saber que a chamada a *read_ahead* é executada *depois* que a resposta é enviada, de modo que o usuário será capaz de continuar executando mesmo se o sistema de arquivos precisar esperar um bloco de disco enquanto faz leitura adiante.

Read_ahead em si é projetado somente para pedir mais um bloco. Ele chama a última função em *read.c*, *rahead*, para realmente fazer o trabalho. *Rahead* (linha 23805) trabalha de acordo com a teoria de que se um pouco a mais é bom, um monte a mais é melhor ainda. Uma vez que discos e outros dispositivos de armazenamento freqüentemente levam um tempo relativamente longo para localizar o primeiro bloco solicitado, mas, então, podem ler de modo relativamente rápido um número de blocos adjacentes, é possível obter muitos mais blocos lidos com um pequeno esforço adicional. Uma solicitação de pré-busca é feita para *get_block*, que prepara o *cache* de blocos para receber diversos blocos de uma vez. Então, *rw_scattered* é chamado com uma lista de blocos. Já discutimos isso antes; lembre-se de que quando os *drivers* de dispositivo são realmente chamados por *rw_scattered*, cada um é livre para responder somente à quantidade de solicitações que ele pode tratar de maneira eficiente. Tudo isso parece complicado, mas as complicações tornam possível uma aceleração significativa em aplicativos que lêem grandes quantidades de dados do disco.

A Figura 5-40 mostra as relações entre alguns procedimentos importantes envolvidos na leitura de um arquivo, em particular, quem chama quem.

Gravando um Arquivo

O código para gravar arquivos está em *write.c*. A gravação de um arquivo é semelhante à leitura, e *do_write* (linha 24025) simplesmente chama *read_write* com o sinalizador *WRITING*. Uma diferença importante entre leitura e gravação é que a gravação requer alocar blocos de disco novos. *Write_map* (linha 24036) é análogo a *read_map*, exceto que em vez de pesquisar números físicos de bloco no nó-i e seus blocos indiretos, ele insere novos aí (para ser preciso, ele insere números de zona, não números de bloco).

O código de *write_map* é longo e detalhado porque deve lidar com vários casos. Se a zona a ser inserida estiver perto do começo do arquivo, ela simplesmente insere no nó-i (linha 24058).

O pior caso é quando um arquivo excede o tamanho que pode ser tratado por um bloco indireto simples, de modo que um bloco indireto duplo agora é solicitado. Em seguida, um bloco indireto simples deve ser alocado, e seu endereço colocado no bloco indireto duplo. Como com a leitura, um procedimento separado, *wr_indir*, é chamado. Se o bloco indireto duplo for adquirido corretamente, mas o disco estiver cheio, então, o bloco indireto simples não pode ser alocado; sendo assim, o duplo deve ser devolvido para evitar corromper o mapa de bits.

Novamente, se pudéssemos apenas desistir e aceitar a pane nesse ponto, o código seria muito mais simples. Entretanto, do ponto de vista do usuário, é muito mais amigável que o esgotamento do espaço em disco simplesmente retornasse um erro de WRITE, em vez de "travar" o computador com um sistema de arquivos corrompido.

Wr_indir (linha 24127) chama uma das rotinas de conversão, *conv2* ou *conv4* para fazer qualquer conversão de dados necessária e coloca um novo número de zona em um bloco indireto. Lembre-se de que o nome dessa função, como os nomes de muitas outras funções que envolvem leitura e gravação, não é literalmente verdadeiro. A gravação real para o disco é tratada pelas funções que mantêm o *cache* de blocos.

O próximo procedimento em *write.c* é *clear_zone* (linha 24149), que cuida do problema de apagar blocos que estão repentinamente no meio de um arquivo. Isso acontece quando uma busca é feita além do fim de um arquivo, seguida por uma gravação de alguns dados. Felizmente, essa situação não ocorre com muita freqüência

New_block (linha 24190) é chamado por *rw_chunk* sempre que um novo bloco é necessário. A Figura 5-41

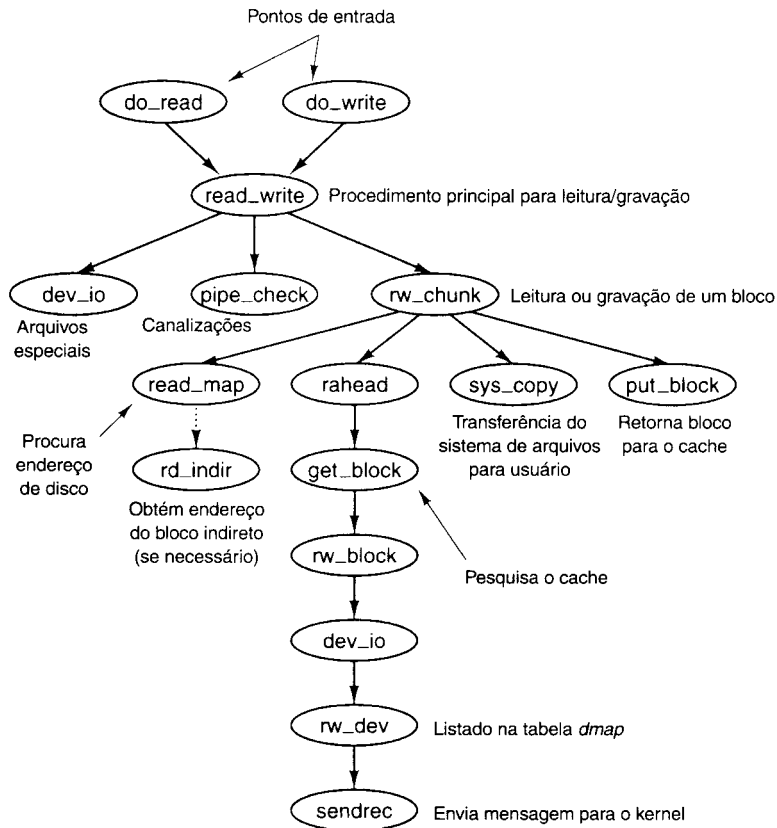



Figura 5-40 Alguns procedimentos envolvidos na leitura de um arquivo.

mostra seis etapas sucessivas do crescimento de um arquivo seqüencial. O tamanho de bloco é de 1K e o tamanho de zona é de 2K nesse exemplo.

Da primeira vez que *new_block* é chamado, ele aloca a zona 12 (blocos 24 e 25). Da próxima vez, ele utiliza o bloco 25, que já foi alocado, mas não está ainda em utilização. Na terceira chamada, a zona 20 (blocos 40 e 41) é alocada e assim por diante. *Zero_block* (linha 24243) limpa um bloco, apagando seu conteúdo anterior. Essa descrição é consideravelmente mais longa do que o código real.

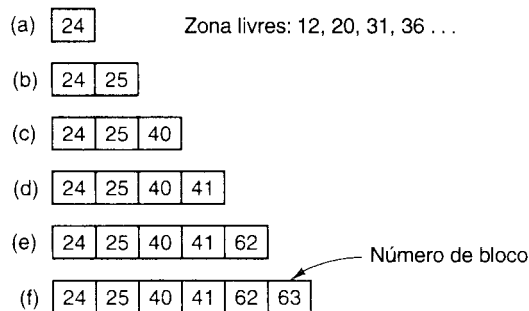



Figura 5-41 (a)-(f) Alocação sucessiva de blocos de 1K com uma zona de 2K.

Canalizações (Pipe)

As canalizações são semelhantes a arquivos comuns sob muitos aspectos. Nessa seção, focalizaremos as diferenças. O código que discutiremos está todo em *pipe.c*

Antes de tudo, as canalizações são criadas de maneiras diferentes, pela chamada PIPE, em vez da chamada CREAT. A chamada PIPE é tratada por *do_pipe* (linha 24332). Tudo que *do_pipe* realmente faz é alocar um nó-i para a canalização e retornar dois descritores de arquivo para ele. As canalizações são possuídas pelo sistema, não pelo usuário, e estão localizadas no dispositivo de canalização designado (configurado em *include/minix/config.h*), que poderia muito bem ser um disco de RAM, uma vez que dados de uma canalização não precisam ser conservados permanentemente.

Ler e gravar uma canalização é ligeiramente diferente de ler e gravar um arquivo, porque um *pipe* tem uma capacidade finita. Uma tentativa de gravar em uma canalização que já está cheia fará com que o gravador seja suspenso. De maneira semelhante, a leitura de uma canalização vazia suspenderá o leitor. Com efeito, uma canalização tem dois ponteiros, a posição atual (utilizado por leitores) e o tamanho (utilizado por escritores), para determinar a origem e o destino dos dados.

As diversas verificações para ver se uma operação em uma canalização é possível são executadas por *pipe_check* (linha 24385). Além dos testes anteriores, que podem levar o chamador a ser suspenso, *pipe_check* chama *release* para ver se um processo previamente suspenso, devido a nenhum dado ou a dados demais, agora pode ser reanimado. Essas reanimações são feitas nas linhas 24413 a 24452, para escritores e leitores que estão dormindo, respectivamente. A gravação em uma canalização quebrada (nenhum leitor) também é detectada aqui.

O ato de suspender um processo é feito por *suspend* (linha 24463). Tudo que ele faz é salvar os parâmetros da chamada na tabela de processos e configurar o sinalizador *dont_reply* como *TRUE*, inibindo a mensagem de resposta do sistema de arquivos.

O procedimento *release* (linha 24490) é chamado para verificar se um processo que foi suspenso em uma canalização pode agora ter permissão para continuar. Se encontrar um, ele chama *revive* para configurar um sinalizador de modo que o laço principal o note mais tarde. Essa função não é uma chamada de sistema, mas é listada na Figura 5-27(c), utilizando o mecanismo de passagem de mensagem.

O último procedimento em *pipe.c* é *do_unpause* (linha 24560). Quando o gerenciador de memória está tentando sinalizar um processo, ele deve saber se esse processo está pendurado em uma canalização ou em um arquivo especial (caso em que deve ser acordado com um erro *EINTR*). Uma vez que o gerenciador de memória não sabe nada sobre canalizações nem sobre arquivos especiais, ele envia uma mensagem para o sistema de arquivos para solicitar. Essa mensagem é processada por *do_unpause*, que reanima o processo se ele estiver bloqueado. Como *revive*, *do_unpause* tem alguma semelhança com uma chamada de sistema, embora não seja uma.

5.7.5 Diretórios e Caminhos

Agora terminamos de ver como arquivos são lidos e gravados. Nossa próxima tarefa é ver como nomes de caminho e diretórios são tratados.

Convertendo um Caminho em um Nó-i

Muitas chamadas de sistema (p. ex., OPEN, UNLINK e MOUNT) têm nomes de caminho (i. e., nomes de arquivo) como parâmetro. A maioria dessas chamadas deve buscar o nó-i para o arquivo nomeado antes de poderem começar a trabalhar na própria chamada. A maneira como um nome de caminho é convertido em um nó-i é um assunto que agora veremos detalhadamente. Já vimos um esboço geral na Figura 5-14.

A análise sintática de nomes de caminho é feita no arquivo *path.c*. O primeiro procedimento, *eat_path* (linha 24727), aceita um ponteiro para um nome de caminho, analisa-o sintaticamente, arranja para seu nó-i ser carregado na memória e retorna um ponteiro para o nó-i. Ele faz seu trabalho chamando *last_dir* para obter o nó-i para o diretório final e, então, chama *advance* para obter o componente final do caminho. Se a pesquisa falhar, por exemplo, porque um dos diretórios ao longo do caminho não existe, ou existe, mas está protegido contra pesquisa, *NIL_INODE* é retornado em vez de um ponteiro para o nó-i.

Os nomes de caminho podem ser absolutos ou relativos e podem ter muitos componentes arbitrariamente separados por barras. Essas questões são tratadas por *last_dir* (linha 24754). Este último começa (linha 24771) examinando o primeiro caractere do nome de caminho para ver se é um caminho absoluto ou relativo. Para caminhos absolutos, *rip* é configurado para apontar para o nó-i raiz; para relativos, é configurado para apontar para o nó-i do diretório de trabalho atual.

Nesse ponto, *last_dir* tem o nome de caminho e um ponteiro para o nó-i do diretório em que pesquisar o primeiro componente. Ele entra em um laço na linha 24782 agora, analisa sintaticamente o nome de caminho, componente por componente. Quando chega ao fim, retorna um ponteiro para o diretório final.

Get_name (linha 24813) é um procedimento utilitário que extrai componentes de cadeias de caracteres (*strings*). Mais interessante é *advance* (linha 24855), que toma como parâmetros um ponteiro de diretório e uma cadeia e pesquisa a cadeia no diretório. Se localizar a *string*, *advance* retorna um ponteiro para seu nó-i. Os detalhes da transferência por intermédio do sistema de arquivos montados são tratados aqui.

Embora *advance* controle a pesquisa de *string*, a comparação real da *string* contra as entradas de diretório é fei-

ta em *search_dir* (linha 24936), que é o único lugar no sistema de arquivos onde arquivos de diretório realmente são examinados. Ele contém dois laços aninhados, um sobre os blocos em um diretório e outro sobre as entradas em um bloco. *Search_dir* também é utilizado para entrar e para excluir nomes de diretórios. A Figura 5-42 mostra os relacionamentos entre alguns procedimentos importantes utilizados ao pesquisar nomes de caminho.

Montando Sistemas de Arquivos

Duas chamadas de sistema que afetam o sistema de arquivos como um todo, MOUNT e UMOUNT, permitem que sistemas de arquivos independentes em dispositivos secundários diferentes "fiquem colados", formando uma única árvore de nomes transparente. A montagem, como vimos na Figura 5-32, é efetivamente alcançada lendo o nó-i raiz e o superbloco do sistema de arquivos a ser montado e configurando dois ponteiros em seu superbloco. Um deles aponta para o nó-i montado e o outro aponta para o nó-i da raiz do sistema de arquivos montado. Esses ponteiros conectam os sistemas de arquivos entre si.

A configuração desses ponteiros é feita no arquivo *mount.c* por *do_mount* nas linhas 25231 e 25232. As duas páginas de código que precedem a configuração dos ponteiros são quase inteiramente dedicadas a verificar todos os erros que podem ocorrer na montagem de um sistema de arquivos, entre eles:

1. O arquivo especial dado não é um dispositivo de blocos.
2. O arquivo especial é um dispositivo de blocos, mas já está montado.
3. O sistema de arquivos a ser montado tem um número mágico podre.
4. O sistema de arquivos a ser montado é inválido (p. ex., nenhum nó-i).
5. O arquivo a ser montado não existe ou é um arquivo especial.
6. Não há espaço para os mapas de bits do sistema de arquivos montado.
7. Não há espaço para o superbloco do sistema de arquivos montado.
8. Não há espaço para o nó-i raiz do sistema de arquivos montado.

Talvez pareça impróprio continuar tocando harpa nesse ponto, mas a realidade de qualquer sistema operacional prático é que uma fração substancial do código é dedicada a fazer tarefas menores que não são intelectualmente muito estimulantes, mas são cruciais para tornar um sistema utilizável. Se um usuário tentar montar o disquete errado acidentalmente, digamos, uma vez por mês e isso levar a uma queda e a um sistema de arquivos corrompido, o usuário irá considerar o sistema como instável e culpará o projetista, não a si próprio.

Thomas Edison uma vez fez uma observação que é relevante aqui. Ele disse que "gênio" é 1% inspiração e 99% transpiração. A diferença entre um bom sistema e um sistema medíocre não é o brilho do algoritmo de agendamento do primeiro, mas sua atenção em fazer os detalhes funcionarem direito.

Desmontar um sistema de arquivos é mais fácil do que montar — há menos coisas que podem dar errado. *Do_umount* (linha 25241) trata disso. A única questão real é certificar-se de que nenhum processo tem qualquer arquivo ou diretório de trabalho abertos no sistema de arquivos a ser removido. Essa verificação é simples e direta: simplesmente varrer a tabela inteira de nós-i para ver se qualquer nó-i na memória pertence ao sistema de arquivos a ser removido (outro que não o nó-i raiz). Se encontrar um, a chamada UMOUNT falha.

O último procedimento em *mount.c* é *name_to_dev* (linha 25299), que pega um nome de caminho de arquivo

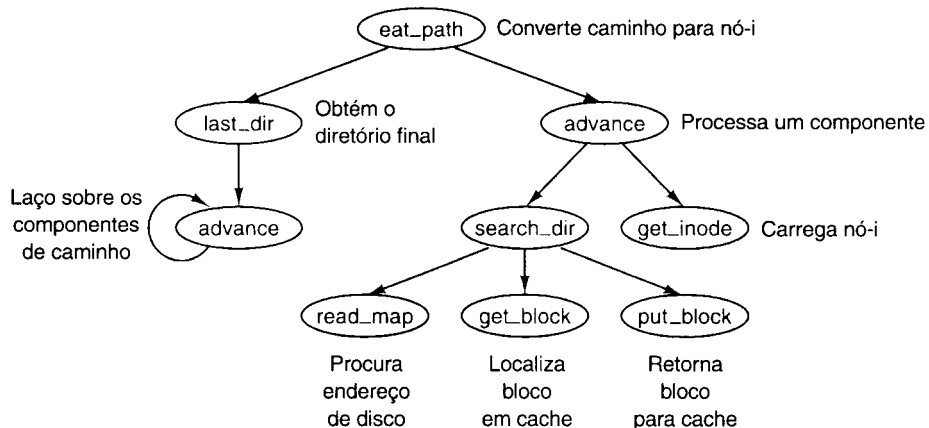



Figura 5-42 Alguns procedimentos utilizados na pesquisa de nomes de caminho.

especial, obtém seu nó-i e extrai seus números de dispositivo primário e secundário. Esses últimos são armazenados no próprio nó-i, no lugar onde a primeira zona normalmente entraria. Essa entrada está disponível porque os arquivos especiais não têm zonas.

Vinculando e Desvinculando Arquivos

O próximo arquivo a considerar é *link.c*, que lida com a vinculação e com a desvinculação de arquivos. O procedimento *do_link* (linha 25434) é muito parecido com *do_mount* sob o aspecto de que quase todo o código é dedicado à verificação de erros. Alguns possíveis erros que podem ocorrer na chamada

link(file_name, link_name);

são listados a seguir:

1. *File_name* não existe ou não pode ser acessado.
2. *File_name* já tem o número máximo de vínculos.
3. *File_name* é um diretório (somente o superusuário pode criar vínculos para ele).
4. *Link_name* já existe.
5. *File_name* e *link_name* estão em dispositivos diferentes.

Se nenhum erro estiver presente, uma nova entrada de diretório é feita com a *string link_name* e o número de nó-i de *file_name*. No código, *name1* corresponde a *file_name* e *name2* corresponde a *link_name*. A entrada real é feita por *search_dir*, chamado de *do_link* na linha 25485.

Arquivos e diretórios são removidos quando desvinculados. O trabalho das chamadas de sistema UNLINK e RMDIR é feito por *do_unlink* (linha 25504). Novamente, diversas verificações devem ser feitas; o teste de ver se um arquivo existe e o teste de ver se um diretório não é um ponto de montagem são feitos pelo código comum em *do_unlink* e, então, *remove_dir* ou *unlink_file* é chamado, dependendo da chamada de sistema que está sendo suportada, o que será discutido em breve.

A outra chamada de sistema suportada em *link.c* é RENAME. Os usuários de UNIX conhecem o comando de *shell mv* que, em última instância, utiliza essa chamada; seu nome reflete outro aspecto da chamada. Não apenas ela pode alterar o nome de um arquivo dentro de um diretório, como também efetivamente pode mover o arquivo de um diretório para outro, e essa chamada pode fazer isso atomicamente, o que previne certas condições de corrida. O trabalho é feito por *do_rename* (linha 25563). Há muitas condições que devem ser testadas antes que esse comando possa ser completado, entre as quais estão:

1. O arquivo original deve existir (linha 25578).
2. O nome de caminho antigo não deve ser um nome de caminho acima do novo diretório na árvore de diretórios (linhas 25596 a 25613).
3. Nem "." nem ".." são aceitáveis como um nome antigo ou novo (linhas 25618 e 25619).
4. Os dois diretórios-pai devem estar no mesmo dispositivo (linha 25622).
5. Os dois diretórios-pai devem ser graváveis, pesquisáveis e estar em um dispositivo gravável (linhas 25625 e 25626).
6. Nem o nome antigo nem o novo podem ser um diretório com um sistema de arquivos montado por cima.

Há algumas outras condições que devem ser verificadas se o novo nome já existir, mas a principal é que deve ser possível remover o arquivo existente com o novo nome.

No código para *do_rename*, há alguns exemplos de decisões de projeto que foram tomadas para minimizar a possibilidade de certos problemas. Renomear um arquivo para um nome que já existe poderia resultar em um disco cheio, mesmo que esse não fosse o caso, se o arquivo antigo não for removido primeiro, e é isso que é feito, nas linhas 25660 a 25666. A mesma lógica é utilizada na linha 25680: remover o nome antigo de arquivo antes de criar um novo nome no mesmo diretório, para evitar a possibilidade de o diretório precisar adquirir um bloco adicional. Entretanto, se fosse o caso de o novo arquivo e de o arquivo antigo estarem em diretórios diferentes, essa preocupação não seria relevante, e na linha 25685 um novo nome de arquivo é criado (em um diretório diferente) antes de o antigo ser removido, porque de um ponto de vista da integridade de sistema uma queda que deixasse dois nomes de arquivo apontando para um nó-i seria muito menos séria que uma queda que deixasse um nó-i não sendo apontado por nenhuma entrada de diretório. A probabilidade de esgotamento do espaço durante uma operação de renomear é baixa, e a de uma queda de sistema mais baixa ainda, mas nesses casos não custa muito estar preparado para o pior caso.

As demais funções em *link.c* suportam as que já discutimos. Para complementar, a primeira delas, *truncate* (linha 25717), é chamada de várias outras posições no sistema de arquivos. Ela passa por um nó-i uma zona por vez, liberando todas as zonas que encontra, assim como os blocos indiretos. *Remove_dir* (linha 25777) executa alguns testes adicionais para assegurar que o diretório pode ser removido e, então, chama *unlink_file* (linha 25818). Se nenhum erro for encontrado, a entrada de diretório é limpa, e a contagem de vínculos no nó-i é diminuída de um.

5.7.6 Outras Chamadas de Sistema

O último grupo de chamadas de sistema é uma mistura de coisas envolvendo status, diretórios, proteção, tempo e outros serviços.

Alterando o Status de Diretórios e de Arquivos

O arquivo *stadir.c* contém o código para quatro chamadas de sistema: CHDIR, CHROOT, STAT e FSTAT. Ao estudar *last_dir* vimos como pesquisas de caminho iniciam olhan-

do no primeiro caractere do caminho, para ver se é uma barra ou não. Dependendo do resultado, um ponteiro, então, é configurado como o diretório de trabalho ou como o diretório-raiz.

A alteração de um diretório de trabalho (ou diretório-raiz) para outro é uma simples questão de alterar esses dois ponteiros dentro da tabela de processos do chamador. Essas alterações são feitas por *do_chdir* (linha 25924) e por *do_chroot* (linha 25963). Essas duas fazem a verificação necessária e, então, chamam *change* (linha 25978) para abrir o novo diretório e substituir o antigo.

Em *do_chdir*, o código nas linhas 25935 a 25951 não é executado em chamadas CHDIR feitas por processos de usuário. Ele é feito especificamente para chamadas do gerenciador de memória, a fim de alternar para um diretório do usuário com o propósito de tratar as chamadas EXEC. Quando um usuário tenta executar um arquivo, digamos, *a.out* em seu diretório de trabalho, é mais fácil para o gerenciador de memória mudar para esse diretório que tentar descobrir onde está.

As duas chamadas de sistema restantes tratadas nesse arquivo, STAT e FSTAT, são basicamente as mesmas, exceto por como o arquivo é especificado. A primeira fornece um nome de caminho, enquanto a última oferece o descritor de arquivo de um arquivo aberto. Os dois procedimentos de primeiro nível, *do_stat* (linha 26014) e *do_fstat* (linha 26035), chamam *stat_inode* para fazer o trabalho. Antes de chamar *stat_inode*, *do_stat* abre o arquivo para obter seu nó-i. Dessa maneira, tanto *do_stat* como *do_fstat* passam um ponteiro de nó-i para *stat_inode*.

Tudo que *stat_inode* (linha 26051) faz é extrair as informações do nó-i e copiar para um buffer, o qual explicitamente deve ser copiado para o espaço do usuário chamando *sys_copy* na linha 26088 porque é muito grande para caber em uma mensagem.

Proteção

O mecanismo de proteção do MINIX utiliza os bits *rwx*. Três conjuntos de bits estão presentes para cada arquivo: para o proprietário, para seu grupo e para outros. Os bits são configurados pela chamada de sistema CHMOD, que é executada por *do_chmod* no arquivo *protect.c* (linha 26124). Depois de fazer uma série de verificações de validade, o modo é alterado na linha 26150.

A chamada de sistema CHOWN é semelhante a CHMOD no sentido de que ambas alteram um campo interno de nó-i em algum arquivo. A implementação também é semelhante embora *do_chown* (linha 26163) possa ser utilizada de modo que o proprietário seja alterado somente pelo superusuário. Usuários comuns podem utilizar essa chamada para alterar o grupo de seus próprios arquivos.

A chamada de sistema UMASK permite que o usuário configure uma máscara (armazenada na tabela de processos), que, então, mascara bits em chamadas de sistema CREAT subseqüentes. A implementação completa seria somente uma declaração, na linha 26209, exceto que a chamada deve retornar o valor antigo de máscara como seu resultado. Esse fardo adicional triplicou o número de linhas de código exigido (linhas 26208 a 26210).

A chamada de sistema ACCESS torna possível para um processo saber se pode acessar um arquivo de uma maneira especificada (p. ex., para leitura). Ela é implementada por *do_access* (linha 26217), que busca o nó-i do arquivo e chama o procedimento interno, *forbidden* (linha 26242), para ver se o acesso é proibido. *Forbidden* verifica o *uid* e o *gid*, assim como as informações no nó-i. Dependendo do que encontra, seleciona um dos três grupos *rwx* e verifica se o acesso é permitido ou proibido.

Read_only (linha 26304) é um procedimento interno pequeno que diz se o sistema de arquivos em que o nó-i passado como parâmetro está montado somente para leitura ou para gravação. É necessário impedir gravações em sistemas de arquivos montados somente para leitura.

Tempo

O MINIX tem várias chamadas de sistema que envolvem tempo: UTIME, TIME, STIME e TIMES. Estão resumidas na Figura 5-43. Embora a maioria delas não tenha nada a ver com arquivos, faz sentido incluí-las no sistema de arquivos porque as informações de tempo são registradas em um nó-i do arquivo.

Associado com cada arquivo estão três números de 32 bits. Dois desses registram os tempos do último acesso e da última modificação do arquivo. O terceiro registra quando o status do próprio nó-i foi por último alterado. Esse tempo irá mudar para quase todos os acessos a um arquivo, exceto um READ ou um EXEC. Esses tempos são mantidos no nó-i. Com a chamada de sistema UTIME, os tempos de

Chamada	Função
UTIME	Configura o tempo da última modificação do arquivo
TIME	Configura o tempo real atual em segundos
STIME	Configura o relógio de tempo real
TIMES	Obtém os tempos de contabilidade de processo

Figura 5-43 As quatro chamadas de sistema envolvendo tempo.

acesso e de modificação podem ser configurados pelo proprietário do arquivo ou pelo superusuário. O procedimento *do_utime* (linha 26422) no arquivo *time.c* executa a chamada de sistema, buscando o nó-i e armazenando o tempo nele. Na linha 26450, os sinalizadores indicando que uma atualização de tempo é necessária são redefinidos, então, o sistema não fará uma cara e redundante chamada a *clock_time*.

O tempo real não é mantido pelo sistema de arquivos. Ele é mantido pela tarefa de relógio dentro do *kernel*. Conseqüentemente, a única maneira de obter ou de configurar o tempo real é enviar uma mensagem à tarefa de relógio. Isso é, de fato, o que fazem *do_time* e *do_stime*. O tempo real está em segundos, desde 1° de janeiro de 1970.

As informações de contabilidade também são mantidas pelo *kernel*. A cada tique de relógio, ele cobra um tique de algum processo. Essas informações podem ser recuperadas enviando uma mensagem à tarefa de sistema, que é o que *do_tims* (linha 26492) faz. O procedimento não é nomeado *do_times* porque a maioria dos compiladores C adiciona um sublinhado à frente de todos símbolos externos, e a maioria dos *linkeditors* trunca símbolos para oito caracteres, o que tornaria *do_time* indistinguível de *do_times*.

Sobras

O arquivo *misc.c* contém procedimentos para algumas chamadas de sistema que não se ajustam em nenhum outro lugar. A chamada de sistema DUP duplica um descritor de arquivo. Em outras palavras, ela cria um novo descritor de arquivo que aponta para o mesmo arquivo que seu argumento. A chamada tem uma variante DUP2. Ambas as versões da chamada são tratadas por *do_dup* (linha 26632). Essa função é incluída no MINIX para suportar programas binários antigos. Essas duas chamadas estão obsoletas. A versão atual da biblioteca C do MINIX invocará a chamada de sistema FCNTL quando qualquer uma dessas for encontrada em um arquivo fonte em C.

FCNTL, tratada por *do_fcntl* (linha 26670) é a maneira preferida de solicitar operações em um arquivo aberto. Os serviços são solicitados utilizando sinalizadores definidos pelo POSIX, descritos na Figura 5-44. A chamada é invocada com um descritor de arquivo, com um código de solicitação e com argumentos adicionais conforme necessário para a solicitação em particular. Por exemplo, o equivalente da antiga chamada

```
dup2(fd, fd2);
```

seria

```
fcntl(fd, F_DUPFD, fd2);
```

Várias dessas solicitações configuram ou lêem um sinalizador; o código consiste em somente algumas linhas. Por exemplo, a solicitação *F_SETFD* configura um bit que força o fechamento de um arquivo quando o processo do seu proprietário faz um EXEC. A solicitação *F_GETFD* é utilizada para determinar se um arquivo deve ser fechado quando uma chamada EXEC é feita. As solicitações *F_SETFL* e *F_GETFL* permitem configurar sinalizadores para indicar que um arquivo particular está disponível no modo não-bloqueador ou para operações *append*.

Do_fcntl também lida com gerenciamento de bloqueio. Uma chamada com o comando *F_GETLK*, *F_SETLK* ou *F_SETLKW* especificada é traduzida em uma chamada a *lock_op*, discutida em uma seção anterior.

A próxima chamada de sistema é SYNC, que copia de volta para o disco todos os blocos e os nós-i que foram modificados desde que foram carregados. A chamada é processada por *do_sync* (linha 26730). Ela simplesmente pesquisa por todas as tabelas, pesquisando entradas sujas. Os nós-i devem ser processados primeiro, uma vez que *rw_inode* deixa seus resultados no *cache* de blocos. Afinal de contas, nós-i sujos são gravados no *cache* de blocos, então, todos os blocos sujos são gravados no disco.

As chamadas de sistema FORK, EXEC, EXIT e SET são, na realidade, chamadas do gerenciador de memória, mas os resultados devem ser colocados aqui também. Quando um processo bifurca, é essencial que o *kernel*, o gerenciador de memória e o sistema de arquivos saibam disso. Essas "chamadas de sistema" não vêm de processos de usuário, mas do gerenciador de memória. *Do_fork*, *do_exit* e *do_set* registram as informações relevantes na parte do sistema de

Operação	Significado
F_DUPFD	Duplica um descritor de arquivo
F_GETFD	Obtém o sinalizador close-on-exec
F_SETFD	Configura o sinalizador close-on-exec
F_GETFL	Obtém sinalizadores de status de arquivo
F_SETFL	Configura sinalizadores de status de arquivo
F_GETLK	Obtém status de bloqueio de um arquivo
F_SETLK	Configura bloqueio de leitura/gravação em um arquivo
F_SETLKW	Configura bloqueio de gravação em um arquivo

Figura 5-44 O POSIX exige parâmetros para a chamada de sistema FCNTL.

arquivos da tabela de processo. *Do_exec* pesquisa e fecha (utilizando *do_close*) qualquer arquivo marcado para ser fechado ao executar (*close-on-exec*).

A última função nesse arquivo não é realmente uma chamada de sistema, mas é tratada como tal. Trata-se de *do_revive* (linha 26921), que é chamada quando uma tarefa que era previamente incapaz de completar o trabalho que o sistema de arquivos tinha solicitado, como oferecer dados de entrada para um processo de usuário, agora completou o trabalho. O sistema de arquivos, então, reanima o processo e envia a mensagem de resposta.

5.7.7 Interface de Dispositivos de E/S

A E/S no MINIX é feita enviando mensagens às tarefas dentro do *kernel*. A interface do sistema de arquivos com essas tarefas está contida no arquivo *device.c*. Quando E/S real de dispositivo é necessária, *dev_io* (linha 27033) é chamada a partir de *read_write* para tratar arquivos especiais de caractere e de *rw_block* para arquivos especiais de bloco. Ela constrói uma mensagem-padrão (veja Figura 3-15) e envia para a tarefa especificada. As tarefas são chamadas pela linha

```
(*dmap[major].dmap-rw)(task, &dev_mess);
```

(linha 27056). Essa chamada funciona via ponteiros na matriz *dmap* definida em *table.c*. As funções que cuidam disso estão todas em *device.c*. Enquanto *dev_io* está esperando uma resposta da tarefa, o sistema de arquivos espera. Ele não tem multiprogramação interna. Normalmente, contudo, essas esperas são bastante curtas (p. ex., 50ms).

Arquivos especiais podem necessitar de processamento especial quando são abertos ou fechados. O que deve ser feito exatamente depende do tipo de dispositivo. A tabela *dmap* também é utilizada para determinar quais funções são indicadas para abrir e fechar cada tipo de dispositivo principal. O procedimento *dev_opcl* (linha 27071) é chamado para dispositivos de disco, sejam disquetes, discos rígidos ou dispositivos baseados em memória. A linha

```
mess_ptr->PROC_NR = fp – fproc;
```

(linha 27081) calcula o número de processo do chamador. O trabalho real é feito passando o número da tarefa e um ponteiro para mensagem a *call_task*, que discutiremos a seguir. *Dev_opcl* também é utilizado para fechar os mesmos dispositivos. De fato, a única diferença entre as funções de abrir e fechar no nível dessa função está no que acontece depois do retorno de *call_task*.

Outras funções chamadas via estrutura *dmap* incluem *tty_open* e *tty_close*, que servem as linhas seriais e *ctty_open* e *ctty_close* que servem o console. O último destes, *ctty_close*, é particularmente uma rotina fictícia, uma vez que tudo que faz é retornar o status de *OK* incondicionalmente.

A chamada de sistema SETSID requer algum trabalho por parte do sistema de arquivos e isso é realizado por *do_setsid* (linha 27164). Uma chamada de sistema, IOCTL, é tratada principalmente em *device.c*. Essa chamada foi colocada aqui porque está intimamente relacionada com a interface de tarefas. Quando uma IOCTL é feita, *do_ioctl* é chamado para construir uma mensagem e enviar para a tarefa adequada.

Para controlar dispositivos de terminal, uma das funções declaradas em *include/termios.h* deve ser utilizada em programas escritos para serem compatíveis com POSIX. A biblioteca C traduzirá tais funções em chamadas IOCTL. Para outros dispositivos que não terminais, IOCTL é utilizada para muitas operações, muitas das quais foram descritas no Capítulo 3.

A próxima função é a única função *PRIVATE* nesse arquivo. Trata-se de *find_dev* (linha 27228), um pequeno procedimento auxiliar que extrai os números de dispositivo primário e secundário de um número de dispositivo completo.

A leitura e a gravação reais da maioria dos dispositivos passa por *call_task* (linha 27245), que dirige uma mensagem à tarefa apropriada na imagem do *kernel*, chamando *sendrec*. A tentativa pode falhar se a tarefa estiver tentando reanimar um processo em resposta a uma solicitação anterior. Provavelmente seria um processo diferente daquele em favor do qual a solicitação atual está sendo feita. *Call_task* exibirá uma mensagem no console se uma mensagem imprópria for recebida. Essas mensagens não serão vistas, esperamos, durante a operação normal do MINIX, mas poderiam aparecer durante tentativas de desenvolver um novo *driver* de dispositivo.

O dispositivo */dev/tty* fisicamente não existe. É uma ficção a que qualquer usuário de sistema *multiusuário* pode referir-se, sem precisar determinar quais de todos os possíveis terminais reais estão em utilização. Quando uma mensagem que referencia */dev/tty* deve ser enviada, a próxima função, *call_ctty* (linha 27311), localiza os dispositivos primário e secundário corretos e substitui-os na mensagem antes de passar a mensagem via *call_task*.

Por fim, a última função no arquivo é *no_dev* (linha 27337), que é chamada para entradas na tabela para as quais um dispositivo não existe, por exemplo, quando um dispositivo de rede é referenciado em uma máquina sem suporte de rede. Ela retorna um status *ENODEV*. Isso previne quedas quando dispositivos inexistentes são acessados.

5.7.8 Utilitários Gerais

O sistema de arquivos contém alguns procedimentos utilitários de propósito geral que são utilizados em vários lugares. Eles estão agrupados no arquivo *utility.c*.

O primeiro procedimento é *clock_time* (linha 27428). Ele envia mensagens à tarefa de relógio para saber qual é o tempo real atual. O próximo procedimento, *fetch_name* (linha 27447), é necessário porque muitas chamadas de sistema têm um nome de arquivo como parâmetro. Se o nome de arquivo for curto, ele será incluído na mensagem do usuário para o sistema de arquivos. Se for longo, um ponteiro para o nome em espaço do usuário será colocado

na mensagem. *Fetch_name* verifica ambos os casos e, de qualquer maneira, obtém o nome.

Duas funções aqui tratam classes gerais de erros. *No_sys* é o manipulador de erro que é chamado quando o sistema de arquivos recebe uma chamada de sistema que não é uma de suas chamadas. *Panic* imprime uma mensagem e diz ao *kernel* para jogar a toalha quando algo catastrófico acontece.

As últimas duas funções, *conv2* e *conv4*, existem para ajudar o MINIX a lidar com o problema de diferentes ordens de byte em processadores Intel e Motorola. Essas rotinas são chamadas durante a leitura ou a gravação para uma estrutura de dados em disco, como um nó-i ou como um mapa de bits. A ordem de byte no sistema que criou o disco é registrada no superbloco. Se é diferente da ordem utilizada pelo processador local, a ordem será trocada. O restante do sistema de arquivos não precisa saber nada sobre a ordem de byte no disco.

O último arquivo é *putk.c*. Ele contém dois procedimentos, ambos tendo a ver com a impressão de mensagens. Os procedimentos padrão de biblioteca não podem ser utilizados, porque enviam mensagens para o sistema de arquivos. Esses procedimentos enviam mensagens diretamente para a tarefa de terminal. Vimos um par de funções quase idêntico na versão do gerenciador de memória desse arquivo.

5.8 RESUMO

Quando visto de fora, um sistema de arquivos é uma coleção de arquivos e de diretórios, mais as operações sobre os mesmos. Os arquivos podem ser lidos e gravados, os diretórios podem ser criados e destruídos, e os arquivos podem ser movidos de um diretório para outro. Sistemas de arquivos mais modernos suportam um sistema de diretórios hierárquico, nos quais diretórios podem ter subdiretórios *ad infinitum*.

Quando visto do interior, um sistema de arquivos parece bem diferente. Os projetistas de sistema de arquivos precisam preocupar-se com o modo como o armazenamento é alocado e com o modo como o sistema monitora qual bloco que vai com qual arquivo. Também vimos como diferentes sistemas têm diferentes estruturas de diretórios. A confiabilidade e o desempenho do sistema de arquivos também são questões importantes.

A segurança e a proteção são de interesse vital tanto para os usuários do sistema como para projetistas. Discutimos algumas falhas de segurança em outros sistemas e problemas genéricos que muitos sistemas têm. Também vimos autenticação, com e sem senhas, listas de controle de acesso e capacitações, assim como um modelo de matriz para pensar sobre a proteção.

Por fim, estudamos o sistema de arquivos do MINIX detalhadamente. Ele é grande, mas não muito complicado. Ele aceita solicitações de trabalho de processos de usuário, pesquisa em uma tabela de ponteiros de procedimentos e chama o procedimento para executar a chamada de sistema solicitada. Devido à sua estrutura modular e à posição fora do *kernel*, ele pode ser removido do MINIX e utilizado como um servidor de arquivos de rede independente com apenas pequenas modificações.

Internamente, o MINIX *bufferiza* os dados em um *cache* de blocos e tenta fazer leitura antecipada ao fazer acesso seqüencial a arquivo. Se o *cache* tiver sido feito grande o suficiente, a maior parte do texto do programa já estará na memória durante operações que acessem repetidamente um conjunto particular de programas, como uma compilação.

EXERCÍCIOS

1. Forneça cinco nomes diferentes de caminho para o arquivo */etc/passwd*. (Sugestão: pense nas entradas de diretório "." e "..".)

2. Os sistemas que suportam arquivos seqüenciais sempre têm uma operação para retroceder arquivos. Sistemas que suportam acesso aleatório a arquivos precisam disso também?

3. Alguns sistemas operacionais oferecem uma chamada de sistema RENAME para dar um novo nome a um arquivo. Há qualquer diferença entre utilizar essa chamada para renomear um arquivo e simplesmente copiar o arquivo para um novo arquivo com o novo nome, seguido da exclusão do antigo?

4. Considere a árvore de diretórios da Figura 5-7. Se */usr/jim* é o diretório de trabalho, qual é o nome de caminho absoluto para o arquivo cujo nome de caminho relativo é *../ast/x*?

5. A alocação contígua de arquivos leva à fragmentação de disco, como mencionado no texto. Essa fragmentação é interna ou externa? Faça uma analogia com algo discutido no capítulo anterior.

6. Um sistema operacional somente suporta um único diretório, mas permite que o diretório tenha arbitrariamente muitos arquivos com nomes de arquivo arbitrariamente longos. Pode-se simular algo próximo de um sistema de arquivos hierárquico? Como?

7. O espaço livre em disco pode ser monitorado utilizando uma lista de blocos livres ou um mapa de bits. Os endereços de disco requerem D bits. Para um disco com B blocos, F dos quais estão livres, declare a condição sob qual a lista de livres utiliza menos espaço que o mapa de bits. Para D tendo o valor de 16 bits, expresse sua resposta como uma porcentagem do espaço em disco que deve estar livre.

8. Foi sugerido que a primeira parte de cada arquivo UNIX seja mantida no mesmo bloco de disco que seu nó-i. Que benefício isso traria?

9. O desempenho de um sistema de arquivos depende da taxa de acerto de *cache* (fração de blocos localizados no *cache*). Se leva 1ms para satisfazer uma solicitação do *cache*, mas 40ms para satisfazer uma solicitação se uma leitura de disco é necessária, dê uma fórmula para o tempo médio exigido para satisfazer uma solicitação se a taxa de acerto é h. Represente graficamente essa função para valores de h de 0 a 10.

10. Um disquete tem 40 cilindros. Uma busca leva 6ms por cilindro movido. Se nenhuma tentativa for feita para colocar os blocos de um arquivo perto um do outro, dois blocos que são logicamente consecutivos (i. e., um segue o outro no arquivo) terão aproximadamente 13 cilindros de distância entre si, na média. Se, entretanto, o sistema operacional fizer uma tentativa de agrupar os blocos relacionados, a distância média entre blocos pode ser reduzida a dois cilindros (por exemplo). Quanto tempo leva para ler um arquivo de 100 blocos em ambos os casos, se a latência rotacional é de 100ms e o tempo de transferência é de 25ms por bloco?

11. A compactação periódica do armazenamento de disco seria de qualquer valor concebível? Explique.

12. Como o TENEX poderia ser modificado para evitar o problema de senha descrito no texto?

13. Depois de formar-se, você se candidata para um emprego como diretor de um grande CPD de uma universidade que acaba de jogar fora seu sistema operacional antigo e alternar para o UNIX. Você consegue o trabalho. Quinze minutos depois de começar a trabalhar, seu assistente entra em seu escritório e grita: "Alguns alunos descobriram o algoritmo que utilizamos para criptografar senhas e publicaram-no no BBS". O que você deve fazer?

14. O esquema de proteção de Morris-Thompson com números aleatórios de n bits foi projetado para tornar difícil para um intruso descobrir um grande número de senhas por criptografar *strings* comuns de antemão. Esse esquema também oferece proteção contra um aluno usuário que está tentando adivinhar a senha de superusuário na sua máquina?

15. Um departamento de ciência da computação tem uma grande coleção de máquinas UNIX em sua rede local. Os usuários em qualquer máquina podem dar um comando da forma

```
machine4 who
```

e fazer com que ele seja executado em *machine4*, sem precisar ter o usuário conectado na máquina remota. Esse recurso é implementado fazendo o *kernel* do usuário enviar o comando e seu *uid* para a máquina remota. Esse esquema é seguro se os *kernels* são todos confiáveis (p. ex., minicomputadores de tempo compartilhado de grande porte com hardware de proteção)? E se algumas máquinas são computadores pessoais de alunos, sem hardware de proteção?

16. Quando um arquivo é removido, seus blocos geralmente são devolvidos na lista de livres, mas eles não são apagados. Você acha que seria uma boa idéia ter o sistema operacional apagando cada bloco antes de liberá-lo? Considere tanto os fatores tanto segurança como desempenho em sua resposta e explique o efeito de cada um.

17. Três mecanismos diferentes de proteção que discutimos são capacidades, listas de controle de acesso, e os bits *rwx* do UNIX. Para cada um dos seguintes problemas de proteção, diga qual desses mecanismos pode ser utilizado.
   (a) Ken quer que seus arquivos possam ser lidos por todo mundo, exceto por seu companheiro de escritório.
   (b) Mitch e Steve querem compartilhar alguns arquivos secretos.
   (c) Linda quer que alguns dos seus arquivos sejam públicos.

   Para UNIX, suponha que os grupos são categorias como professores e funcionários, alunos, secretárias, etc.

18. Considere o seguinte mecanismo de proteção. A cada objeto e a cada processo é atribuído um número. Um processo somente pode acessar um objeto se o objeto tiver um número mais alto que o processo. Com qual dos esquemas de arquivo discutidos no texto isso se assemelha? De que maneira essencial ele difere do esquema no texto?

19. O cavalo de Tróia pode atacar o trabalho em um sistema protegido por capacidades?

20. Dois alunos de ciência da computação, Carolyn e Elinor, estão tendo uma discussão sobre nós-i. Carolyn sustenta que as memórias ficaram tão grandes e tão baratas que quando um arquivo é aberto, é mais simples e mais rápido simplesmente colocar uma nova cópia do nó-i na tabela de nós-i, em vez de pesquisar a tabela inteira para ver se ele já está lá. Elinor discorda. Quem está certo?

21. Qual é a diferença entre um vírus e um verme? Como cada um deles se reproduz?

22. Vínculos simbólicos são arquivos que apontam para outros arquivos ou para outros diretórios indiretamente. Diferentemente de vínculos comuns como aqueles atualmente implementados no MINIX, um vínculo simbólico tem seu próprio nó-i, que aponta para um bloco de dados. Os bloco de dados contêm o caminho para o arquivo de destino do vínculo, e o nó-i torna possível para o vínculo ter proprietários e permissões diferentes dos do arquivo de destino do vínculo. Um vínculo simbólico e o arquivo ou diretório para que ele aponta podem ser localizados em dispositivos diferentes. Vínculos simbólicos não são parte do padrão POSIX 1990, mas espera-se que sejam adicionados ao POSIX no futuro. Implemente vínculos simbólicos para o MINIX.

23. Você descobre que o limite de tamanho de arquivo de 64MB no MINIX não é suficiente para suas necessidades. Estenda o sistema de arquivos, utilizando o espaço não-utilizado no nó-i para um bloco indireto triplo.

24. Mostre se configurar ROBUST torna o sistema de arquivos mais ou menos robusto em face de uma queda. Se esse é o caso, na versão atual do MINIX ainda não foi explorado, portanto, pode ser qualquer um. Examine bem o que acontece quando um bloco modificado é expulso do *cache*. Leve em conta que blocos de dados modificados podem ser acompanhados por um nó-i e por um mapa de bits modificado.

25. O tamanho da tabela *filp* atualmente está definido como uma constante, *NR_FILPS*, em *fs/const.h*. Para acomodar mais usuários em um sistema em rede, você quer aumentar *NR_PROCS* em *include/minix/config.h*. De que modo *NR_FILPS* deveria ser definido como uma função de *NR_PROCS*?

26. Projete um mecanismo para adicionar suporte a um sistema de arquivos "estrangeiro", de modo que se poderia, por exemplo, montar um sistema de arquivos MS-DOS em um diretório no sistema de arquivos do MINIX.

27. Suponha que um avanço tecnológico ocorra e que a RAM não-volátil, que mantém seu conteúdo confiável após uma queda de energia, torne-se disponível sem desvantagem de preço ou de desempenho sobre a RAM convencional. Quais aspectos do projeto do sistema de arquivos seriam afetados por esse desenvolvimento?

6

Lista de Leitura e Bibliografia

Nos cinco capítulos anteriores, abordamos uma variedade de temas. Este capítulo foi concebido como uma ajuda a leitores interessados em se aprofundar no grande estudo de sistemas operacionais mais ainda. A seção 6.1 é uma lista de leituras sugeridas. A seção 6.2 é uma bibliografia alfabética de todos livros e artigos citados neste livro.

Além das referências dadas a seguir, as *Proceedings of the n-th ACM Symposium on Operating Systems Principles* (ACM) publicadas bianualmente e o *Proceedings of the n-th International Conference on Distributed Computing Systems* (IEEE) publicadas anualmente são bons lugares para procurar papers recentes sobre sistemas operacionais. Assim também é o *Symposium on Operating Systems Design and Implementation* da USENIX. Além disso, os periódicos da ACM *Transactions on Computer Systems* e *Operating Systems Review* freqüentemente têm artigos relevantes.

6.1 SUGESTÕES PARA LEITURAS SUPLEMENTARES

6.1.1 Introdução e Trabalhos Gerais

Brooks, *The Mythical Man-Month: Essays on Software Engineering*

Um livro informativo, divertido e engenhoso sobre como não escrever um sistema operacional, feito por alguém que aprendeu da maneira mais difícil. Repleto de bons conselhos.

Comer, *Operation System Design. The Xinu Approach*

Um livro sobre o sistema operacional Xinu, que roda no computador LSI-11. Contém uma detalhada exposição do código-fonte, incluindo uma listagem completa em C.

Corbató. "On Building Systems That Will Fail"

Esta é a conferência que recebeu o Prêmio Turing, o pai do compartilhamento de tempo aborda muitas das mesmas preocupações que interessam a Brooks em *The Mythical Man-Month*. Sua conclusão é que todos os sistemas complexos finalmente falharão e que, para ter qualquer chance para êxito, é absolutamente essencial evitar a complexidade e esforçar-se para utilizar a simplicidade e a elegância em um projeto.

Deitel, *Operating Systems*, 2ª Ed.

Um texto geral sobre sistemas operacionais. Além do material padrão, contém estudos de caso de UNIX, MS-DOS, MVS, VM, OS/2, e o sistema operacional do Macintosh.

Finkel, *An Operating Systems Vade Mecum*

Outro texto geral sobre sistemas operacionais. É orientado para a prática, bem escrito e cobre muitos dos temas tratados neste livro, tornando-o um bom lugar para procurar uma perspectiva diferente sobre o mesmo assunto.

IEEE, *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Este é o padrão. Algumas partes são realmente bem legíveis, especialmente o Anexo B, "Rationale and Notes",

que freqüentemente joga uma luz sobre por que as coisas são feitas como são. Uma vantagem de consultar o documento-padrão é que, por definição, não há nenhum erro. Se um erro tipográfico em um nome de macro passar pelo processo de edição não é mais um erro, é oficial.

Lampson, "Hints for Computer System Design"
Butler Lampson, um dos principais projetistas do mundo de sistemas operacionais inovadores, colecionou muitas dicas, sugestões e diretrizes de seus anos de experiência e reuniu-os neste artigo que informa e entretém. Como o livro de Brooks, essa é leitura obrigatória para todo aspirante a projetista de sistema operacional.

Lewine, *POSIX Programmer's Guide*
Este livro descreve o padrão POSIX de uma maneira muito mais legível do que os próprios documentos do padrão, e inclui discussões sobre como converter programas mais velhos para POSIX e como desenvolver novos programas para o ambiente POSIX. Há numerosos exemplos de código, incluindo vários programas completos. Todas as funções de biblioteca e arquivos de cabeçalho exigidos pelo POSIX são descritos.

Silberschatz and Galvin, *Operating System Concepts,* 4th Ed.
Outro texto sobre sistemas operacionais. Cobre processos, gerenciamento de armazenamento, de arquivos e de sistemas distribuídos. Dois estudos de caso são dados: UNIX e Mach. O material está repleto de dinossauros. O que isso tem a ver com sistema operacional, se é que tem alguma coisa a ver, em plena década de 90 não se sabe.

Stallings, *Operating Systems*, 2nd Ed.
Ainda outro texto sobre sistemas operacionais. Cobre todos os temas normais e também inclui uma pequena quantia de material sobre sistemas distribuídos, mais um apêndice sobre teoria de filas.

Stevens, *Advanced Programming in the UNIX Environment*
Este livro diz como escrever programas em C que utilizam a interface de chamada de sistema do UNIX e a biblioteca padrão de C. Os exemplos são baseados no System V Release 4 e nas versões 4.4BSD do UNIX. O relacionamento dessas implementações com o POSIX é descrito em detalhes.

Switzer, *Operating Systems, A Practical Approach*
Uma abordagem semelhante à deste texto. Conceitos teóricos são ilustrados com exemplos de pseudocódigo e com uma grande parte do código-fonte em C para o TUNIX, um sistema operacional modelo. Diferentemente do MINIX, o TUNIX não é destinado a executar em uma máquina real, ele executa em uma máquina virtual. Não é tão realista quanto o MINIX em seu tratamento de *drivers* de dispositivo, mas aprofunda-se mais do que o MINIX em outras direções, como a implementação de memória virtual.

6.1.2 Processos

Andrews and Schneider, "Concepts and Notations for Concurrent Programming"
Um tutorial e uma pesquisa sobre processos e comunicação interprocesso, incluindo espera ativa, semáforos, monitores, passagem de mensagens e outras técnicas. O artigo também mostra como esses conceitos são embutidos em várias linguagens de programação.

Ben-Ari, *Principles of Concurrent Programming*
Este pequeno livro é inteiramente dedicado aos problemas de comunicação interprocesso. Há capítulos sobre exclusão mútua, semáforos, monitores e o problema dos filósofos jantando, entre outros.

Dubois e colaboradores, "Synchronization, Coherence, and Event Ordering in Multiprocessors"
Um tutorial sobre sincronização em sistemas multiprocessados com compartilhamento de memória. Algumas idéias, entretanto, também são igualmente aplicáveis a sistemas de processador único e de memória distribuída.

Silberschatz and Galvin. *Operating System Concepts*, 4th Ed.
Os Capítulos 4 a 6 abrangem processos e comunicação interprocesso, incluindo agendamento, seções críticas, semáforos, monitores e problemas clássicos de comunicação interprocesso.

6.1.3 Entrada/Saída

Chen e colaboradores, "RAID: High Performance Reliable Secondary Storage"
O uso de múltiplas unidades de disco em paralelo para E/S rápida é uma tendência em sistemas sofisticados. Os autores discutem essa idéia e examinam diferentes organizações em termos de desempenho, de custo e de confiabilidade.

Coffman e colaboradores, "System Deadlocks"
Uma breve introdução a impasses, o que os causa e como eles podem ser prevenidos ou detectados.

Finkel, *An Operating Systems Vade Mecum*, 2ª Ed.
O Capítulo 5 discute hardware de E/S e *drivers* de dispositivo, particularmente para terminais e discos.

Geist and Daniel, "A Continuum of Disk Scheduling Algorithms"
Um algoritmo generalizado de agendamento de braço de disco é apresentado. Simulação extensa e resultados experimentais são oferecidos.

Holt, "Some Deadlock Properties of Computer Systems"
Uma discussão sobre impasses. Holt introduz um modelo de gráfico dirigido que pode ser utilizado para analisar algumas situações de impasse.

IEEE *Computer* Magazine, Março de 1994
Esse exemplar da IEEE *Computer* contém oito artigos sobre E/S avançada e abrange simulação, armazenamento de alto de desempenho, *cache*, E/S para computadores paralelos e multimídia.

Isloor and Marsland, "The Deadlock Problem: An Overview"
Um tutorial sobre impasses, com ênfase especial sobre sistemas de banco de dados. Uma variedade de modelos e de algoritmos são abordados.

Stevens, "Heuristics for Disk Drive Positioning in 4.3BSD"
Um detalhado estudo sobre desempenho de disco no UNIX da Berkeley. Como é freqüentemente o caso com sistemas de computador, a realidade é mais complicada que o previsto na teoria.

Wilkes e colaboradores, "The HP AutoRAID Hierarchical Storage System"
Um novo desenvolvimento importante em sistemas de disco de alto desempenho é RAID (*Redundant Array of Inexpensive Disks*), no qual uma matriz de discos pequenos funciona para produzir um sistema de alta largura de banda. Neste paper, os autores descrevem com algum detalhe o sistema que eles construíram nos laboratórios da HP.

6.1.4 Gerenciamento de Memória

Denning, "Virtual Memory"
Um paper clássico sobre muitos aspectos da memória virtual. Denning foi um dos pioneiros neste campo e o inventor do conceito de conjunto funcional.

Denning, "Working Sets Past and Present"
Uma boa visão geral de numerosos algoritmos de gerenciamento de memória e de paginação. Uma bibliografia abrangente é incluída.

Knuth, *The Art of Computer Programming* Vol. I
Primeiro ajuste, melhor ajuste e outros algoritmos de gerenciamento de memória são discutidos e comparados neste livro.

Silberschatz and Galvin, *Operating System Concepts*, 4ª Ed.
Os Capítulos 8 e 9 lidam com gerenciamento de memória, incluindo troca, paginação e segmentação. Uma variedade de algoritmos de paginação é mencionada.

6.1.5 Sistemas de Arquivos

Denning, "The United States vs. Craig Neidorf"
Quando um jovem *hacker* descobriu e publicou informações sobre como o sistema telefônico funciona ele foi acusado de fraude de computador. Este artigo descreve o caso, que envolveu muitas questões fundamentais, incluindo liberdade de expressão. O artigo é seguido por alguns pareceres discordantes e por uma refutação de Denning.

Hafner and Markoff, *Cyberpunk*
Três fascinantes contos de jovens *hackers* invadindo computadores pelo mundo são escritos aqui pelo repórter de informática do New York Times, responsável pelo furo de reportagem sobre o verme que assolou a Internet, e sua esposa jornalista.

Harbron, *File Systems*
Um livro sobre projetos de sistemas de arquivos, aplicativos e desempenho. São abordados tanto a estrutura como os algoritmos.

McKusick e colaboradores, "A Fast File System for UNIX"
O sistema de arquivos UNIX foi completamente reimplementado para 4.2 BSD. Esse paper descreve o projeto do novo sistema de arquivos, com ênfase em seu desempenho.

Silberschatz and Galvin *Operating System Concepts*, 4ª Ed.
Os Capítulos 10 e 11 são sobre sistemas de arquivos. Abrangem operações de arquivo, métodos de acesso, consistência de semântica, diretórios e proteção, e implementação, entre outros temas.

Stallings, *Operating Systems*, 2ª Ed.
O Capítulo 14 contém uma relativa quantidade de material sobre o ambiente de segurança, especialmente sobre *hackers*, vírus e outras ameaças.

6.2 BIBLIOGRAFIA ALFABÉTICA

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Fev. 1992.

ANDREWS, G.R., and SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, Março 1983.

BACH, M.J.: *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice Hall, 1987.

BALA, K., KAASHOEK, M.F., WEIHL, W.: "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243-254, 1994.

BAYS, C.: "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Commun. of the ACM*, vol. 20, pp. 191-192, Março 1977.

BEN-ARI, M: *Principles of Concurrent Programming*, Englewood Cliffs, NJ: Prentice-Hall International, 1982.

BRINCH HANSEN, P.: "The Programming Language Concurrent Pascal," *IEEE Trans. on* Software *Engineering*, vol. SE-1, pp. 199-207, Junho 1975.

BROOKS, F. P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*, Anniversary edition, Reading, MA: Addison-Wesley, 1996.

CADOW, H.: *OS/360 Job Control Language*, Englewood Cliffs, NJ: Prentice-Hall, 1970.

CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.: "RAID: High Performance Reliable Storage." *Computing Surveys*, vol. 26, pp. 145-185. Junho 1994.

CHERITON, D.R.: "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Reviews*, vol. 18, pp. 12-20, Out. 1984.

COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, Junho 1971.

COMER, D.: *Operating System Design. The Xinu Approach*, Englewood Cliffs. NJ.: Prentice-Hall, 1984.

CORBATO, F.J.: "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, Junho 1991.

CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C: "An Experimental Time-Sharing System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.

CORBATO, F.J., SALTZER, J.H., and CLINGEN, C.T.: "MULTICS — The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS. pp. 571-583, 1972.

CORBATO, F.J., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.

COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.: "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Out. 1971.

DALEY, R.C., and DENNIS, J.B.: "Virtual Memory, Process, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, Maio 1968.

DEITEL, H.M.: *Operating Systems*, 2nd Ed., Reading, MA: Addison-Wesley, 1990.

DENNING, D.: "The United states vs. Craig Neidorf," *Commun. of the ACM*, vol. 34, pp. 22-43, Março 1991.

DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.

DENNING, P.J.: "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.

DENNING, P.J.: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Set. 1970.

DENNING, P.J.: "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Jan. 1980.

DENNIS, J.B., and VAN HORN, E.C.: "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, Março 1966.

DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.

DIJKSTRA, E.W.: "The Structure of THE Multiprogramming System," *Commun. of the ACM*, vol. 11, pp. 341-346, Maio 1968.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, vol. 21, pp. 9-21, Fev. 1988.

ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J. Jr.: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. of the Fifteenth Symp. on Operation Systems Principles*, ACM, pp. 251-266, 1995.

FABRY, R.S.: "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412. Julho 1974.

FEELEY, M.J., MORGAN, W.E., PGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.: "Implementing Global Memory Management in a Workstation Cluster," *Proc. of the Fifteenth Symp. on Operation Systems Principles*. ACM, pp. 201-212, 1995.

FINKEL, R.A.: *An Operation Systems Vade Mecum*, 2nd Ed., Englewood Cliffs. NJ: Prentice-Hall, 1988.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Out. 1961.

GEIST, R., and DANIEL, S.: "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Fev. 1987.

GOLDEN, D., and PECHURA, M.: "The Structure of Microcomputer File Systems," *Commun. of the ACM*, vol. 29, pp. 222-230, Março 1986.

GRAHAM, R.: "Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T., Set. 1970.

HAFNER, K., and MARKOFF, J.: *Cyberpunk*, New York: Simon and Schuster. 1991.

HARBRON, T.R.: *File Systems*, Englewood Cliffs, NJ: Prentice Hall, 1988.

HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.: "Using threads in Interactive Systems: A Case Study," *Proc. of the Fourteenth Symp. on Operating Systems Principles*, ACM, pp. 94-105, 1993.

HAVENDER, J.W.: "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, pp. 74-84, 1968.

HEBBARD, B. *et al.*: "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Review*, vol. 14, pp. 7-20, Jan. 1980.

HOARE, C.A.R.: "Monitors, An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549-557, Out. 1974; Erratum in *Commun* of the ACM, vol. 18, p. 95, Fev. 1975.

HOLT, R.C: "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179-196, Set. 1972.

HOLT, R.C: *Concurrent Euclid. The UNIX System. and TUNIS*. Reading. MA: Addison-Wesley, 1983.

HUCK, J., and HAYS, J.: "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. Twentieth Annual Int'l Symp. on Computer Arch.*, ACM, pp. 39-50, 1993.

IEEE: *Information technology—Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API) [C Languages]*, New York: Institute of Electrical and Electronics Engineers, Inc., 1990

ISLOOR, S.S., and MARSLAND, T.A.: "The Deadlock Problem: An Overview," *IEEE Computer*. vol. 13. pp. 58-78, Set. 1980.

KERNIGHAN, B.W., and RITCHIE. D.M.: *The C Programming Language*, 2ª Ed., Englewood Cliffs, NJ: Prentice Hall. 1988.

KLEIN. D.V.: "Foiling the Cracker: A Survey of, and Improvements to, Password Security," *Proc. UNIX Security Workshop II*, USENIX, Verão 1990.

KLEINROCK, L.: *Queueing Systems*, Vol. 1. New York: John Wiley, 1975.

KNUTH, D.E.: The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2ª Ed., Reading, MA: Addison-Wesley, 1973.

LAMPSON, B.W.: "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, Maio 1968.

LAMPSON, B.W.: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Out. 1973.

LAMPSON, B.W.: "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Jan. 1984.

LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.: "Policy/Mechanism Separation in Hydra," *Proc. of the Fifth Symp. on Operating Systems Principles*, ACM, pp. 132-140, 1975.

LEWINE, D.: POSIX *Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.

LINDE, R.R.: "Operating System Penetration," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361-368, 1975.

LIONS, J.: *Lion's Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.

LIU, C.L., and LAYLAND, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46-61, Jan. 1973.

MARSH, B.D., SCOTT, M.L, LEBLANC, T.J., and MARKATOS, E.P.: "First-Class User-Level threads," *Proc. of the Thirteenth Symp. On Operation Systems Principles*, ACM, pp. 110-171, 1991.

McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Agosto 1984.

MORRIS, R., and THOMPSON, K.: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Nov. 1979.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software–Practice and Experience*, vol. 14, pp. 365-368, Abril 1984.

ORGANICK, E.I.: *The Multics System*. Cambridge, MA: M.I.T Press, 1972.

PETERSON, G.L.: "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, Junho 1981.

ROSENBLUM, M., and OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. of the Thirteenth Symp. On Operation Systems Principles*, ACM, pp. 1-15, 1991.

SALTZER, J.H.: "Protection and Control of Information Sharing in MULTICS," *Commun. of the ACM*, vol. 17, pp. 388-402, Julho 1974.

SALTZER, J.H., and SCHROEDER, M.D.: "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Set. 1975.

SALUS, P.H.: "UNIX At 25," *Byte*, vol. 19, pp. 75-82, Out. 1994.

SANDHU,R.S.: "Lattice-Based Access Control Models," *Computer*, vol. 26, pp. 9-19, Nov. 1993.

SEAWRIGHT, L.H., and MACKINNON, R.A.: "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems Journal*, vol. 18, pp. 4-17, 1979.

SILBERSCHATZ, A., and GALVIN, P.B.: *Operating System Concepts*, 4th Ed. Reading, MA: Addison-Wesley, 1994.

STALLINGS, W.: *Operating Systems*, 2ª Ed., Englewood Cliffs, NJ: Prentice-Hall, 1995.

STEVENS, W.R.: *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.

STEVENS, W.R.: "Heuristics for Disk Drive Partitioning in 4.3BSD," *Computing Systems*, vol. 2, pp. 251-274, Verão 1989.

STOLL, C.: The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage, New York: Doubleday, 1989.

SWITZER, R.W.: *Operating Systems, A Practical Approach*, London: Prentice-Hall Int'l, 1993.

TAI, K.C., and CARVER, R.H.: "VP: A New Operation for Semaphores," Operating Systems Review, vol. 30, pp. 5-11, Julho 1996.

TALLURI, M., and HILL, M.D.: "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. on Architectural Support for Progr. Lang. and Operating Systems*, ACM, pp. 171-182, 1994.

TALLURI, M., HILL, M.D., and KHALIDI, Y.A.: "A New Page Table for 64-bit Address Spaces," *Proc. of the Fifteenth Symp. on Operating Systems Principles*, ACM, pp. 184-200, 1995.

TANENBAUM, A.S.: *Distribute Operating System*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, C.J., MULLENDER, S.J., JANSEN, J., and ROSSUM, G. VAN: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46-63, Dez. 1990.

TEORY, T.J.: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf., AFIPS*, pp. 1-11, 1972.

THOMPSON, K.: "Unix Implementation," *Bell System Technical Journal*, vol. 57, pp. 1931-1946, Julho-Agosto 1978.

UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., and BROWN, R.: "Design Trade-offs for Software-Managed TLBs," *ACM Trans. on Computer Systems*, vol. 12, pp. 175-205, Agosto 1994.

VAHALIA, U.: *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice-Hall, 1996.

WALDSPURGER, C.A., and WEIHL, W.E.: "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1-12, 1994.

WILKES, J., GOLDING, R., STAELIN, C, and SULLIVAN, T.: "The HP AutoRAID Hierarchical Storage System," *ACM Trans. on Computer System*, vol. 14, pp. 108-136, Fev. 1996.

WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK F.J.: "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. of the ACM*, vol. 17, pp. 337-345, Junho 1974.

ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.: "Software Write Detection for a Distributed Shared Memory," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87-100, 1994.

APÊNDICES